NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

THE DESIGN AND IMPLEMENTATION OF A SEMI-AUTONOMOUS SURF-ZONE ROBOT USING ADVANCED SENSORS AND A COMMON ROBOT OPERATING SYSTEM

by

Steven Halle
Jason Hickle

June 2011

Thesis Co-Advisors: Richard Harkins
Timothy H. Chung

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*
8–6–2011

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From — To)*
7/6/2009–6/17/2011

4. TITLE AND SUBTITLE
The Design and Implementation of a Semi-Autonomous Surf-Zone Robot using Advanced Sensors and a Common Robot Operating System

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

6. AUTHOR(S)
Steven Halle, Jason Hickle

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Navy

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number N/A.

14. ABSTRACT

A semi-autonomous vehicle, MONTe, was designed, modeled and tested for deployment and operation in a surf-zone coastal environment. The MONTe platform was designed to use unique land based locomotion that incorporates wheel-legs(Whegs^{TM}) and a tail. Semi-autonomy was realized with data from onboard sensors and implemented through open source Robot Operating System (ROS), hosted on an Ubuntu Linux based processor. Communications via IEEE 802.11 protocols proved successful for data telemetry in line of site operations. Basic mobility and tail control of the platform was modeled in Working Model 2D. Field tests were successfully conducted to demonstrate mobility and semi-autonomous waypoint navigation. Future developments will look to improve the overall design and test water borne mobility, navigation, and communication.

15. SUBJECT TERMS
Biologically Inspired, Robotics, Robot Operating System, Wheg^{TM}, Autonomous, Amphibious Robot

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON / 19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	131	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

THE DESIGN AND IMPLEMENTATION OF A SEMI-AUTONOMOUS SURF-ZONE ROBOT USING ADVANCED SENSORS AND A COMMON ROBOT OPERATING SYSTEM

Steven Halle
Lieutenant, United States Navy
B.S., University of Illinois, 2004

Jason Hickle
Lieutenant, United States Navy
B.S., U.S. Naval Academy, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2011

Authors: Steven Halle

Jason Hickle

Approved by: Richard Harkins
Thesis Co-Advisor

Timothy H. Chung
Thesis Co-Advisor

Andres Larraza
Chair, Department of Physics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A semi-autonomous vehicle, MONTe, was designed, modeled and tested for deployment and operation in a surf-zone coastal environment. The MONTe platform was designed to use unique land based locomotion that incorporates wheel-legs(Whegs^{TM}) and a tail. Semi-autonomy was realized with data from onboard sensors and implemented through open source Robot Operating System (ROS), hosted on an Ubuntu Linux based processor. Communications via IEEE 802.11 protocols proved successful for data telemetry in line of site operations. Basic mobility and tail control of the platform was modeled in Working Model 2D. Field tests were successfully conducted to demonstrate mobility and semi-autonomous waypoint navigation. Future developments will look to improve the overall design and test water borne mobility, navigation, and communication.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

FROM STEVE

I first want to thank my beautiful girls. My wonderful wife Alissa offered amazing support throughout this project, all while caring for our unborn son. My daughter Erin always greeted me at the door with a smile and a hug. Their endless encouragement, motivation, and love allowed me to complete this project. I want to thank the great friends at BML that helped inspire the name MONTe and Monty Bauer himself, aka “A Machine in the Water”. I am deeply grateful to Mike Slatt for being proactive with his research and putting in countless hours making sure that MONTe was ready for testing (and without Velcro). Lastly I want to thank my advisors for being flexible and helpful throughout the process while demanding excellence.

FROM JASON

No adventure is effective (nor as fun) without a great support structure. So, I send my love to all my family and friends for everything. You all make it worthwhile. I also thank Mike Slatt for all of his hardware magic despite the numerous times I broke MONTe. Finally, I thank the advisors for all of their counsel and for giving me the opportunity to work on a project like this.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

For nearly a decade, the Naval Postgraduate School (NPS), Case Western University and additional collaborators have been involved in a program to develop an autonomous surf-zone robot. In general, unmanned systems provide significant advantages for military operations and commercial applications. Autonomous and remotely operated systems provide advanced capabilities at a lower cost, while not placing personnel in harm's way. There have been recent advancements in unmanned systems that operate solely in one mode, such as land based or water based. Overcoming the transition from one mode to another, such as an amphibious robot, still requires significant research. The environmental conditions and duality associated with the surf-zone presents a unique set of challenges. There have been several previous platforms that have attempted to address the difficulties associated with these harsh conditions. Some of these platforms have provided invaluable insight into complex mobility and autonomy.

Interest for these types of systems that operate in coastal areas and shallow beaches is widespread. Platforms wielding these capabilities can be outfitted with a multitude of sensors to accomplish a spectrum of missions. Tasking may include minesweeping or clearance, terrain or bathymetry surveys, covert reconnaissance and surveillance. As sensor packages become more compact, it is possible to envision equipping such a robot with a chemical detection unit that may be able to search for specific compounds or chemical weapons. The versatility of these platforms justifies the need to research and develop a robust surf-zone robot.

1.1 Background

1.1.1 Previous Designs

Whegs™ (wheel-legs) describes a class of robot that characterizes its locomotion based on a fusion of a wheel design with crawling leg that was inspired by the motion of biological organisms. Case Western Reserve University's Biologically Inspired Robotics Laboratory developed this means of locomotion under Roger Quinn [6]. The idea was based on the high maneuverability of a cockroach and its ability to overcome adverse obstacles. This design concept has been incorporated into several versions of surf-zone robots.

An earlier Whegs™ design was the Dayton Area Graduate Studies Institute (DAGSI) Whegs™

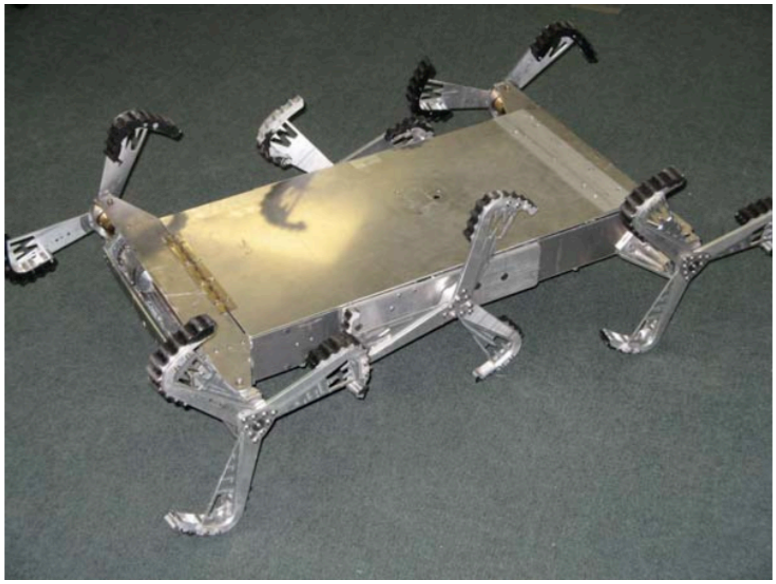

Figure 1.1: From [1], a picture of Agbot

prototype called Agbot. NPS and Case Western Reserve University collaborated to build Agbot, pictured in Figure 1.1. Agbot's design consists of six Whegs^{TM}, a single section body, and is steered by angling the front Whegs^{TM}, similar to an automobile. The autonomy was limited to waypoint navigation from a control station. Agbot's main purpose was to act as a test platform for the mobility of the Whegs^{TM} and their ability to overcome obstacles. Detailed information regarding Agbot is available in [1].

Variants of Agbot have been produced, which have consistent designs and been subject to additional testing [2]. Although success has been shown for climbing large obstacles, improved designs have the potential for significant advances. One proposed improvement is replacing the rear segment of the main body with an autonomous tail. The main motivation for incorporating a tail into a surf-zone robot is to climb larger obstacles and terrain [2]. This modeling was done in Working Model 2D and still needed to be verified with prototype trials. The addition of a tail and removal of two Whegs^{TM} also implied that the stability would be improved (discussed in Section 2.1.1) with a new design comprising four legs per Wheg^{TM} instead of three. A conceptual rendering of a next generation robot is seen in Figure 1.2. For a more detailed explanation of these mobility concepts see Section 2.1.

ROBSTER, Figure 1.3, was developed to serve as a test platform for the addition of a tail [3]. This initial investigation into tail control was conducted by Courtney Holland at NPS in June 2009. ROBSTER's design incorporated the new Whegs^{TM} style and consisted of a rigid tail that

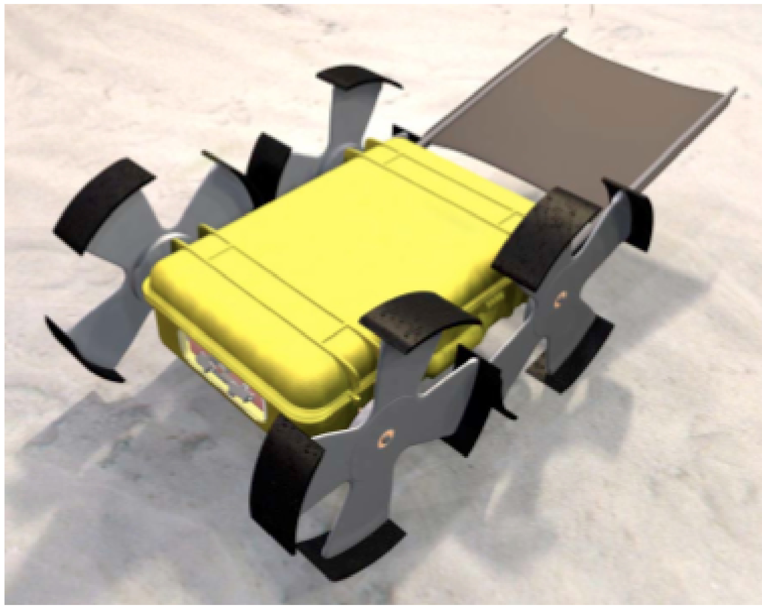

Figure 1.2: From [2], a 3D rendering of a future design called Pelican Whegs™

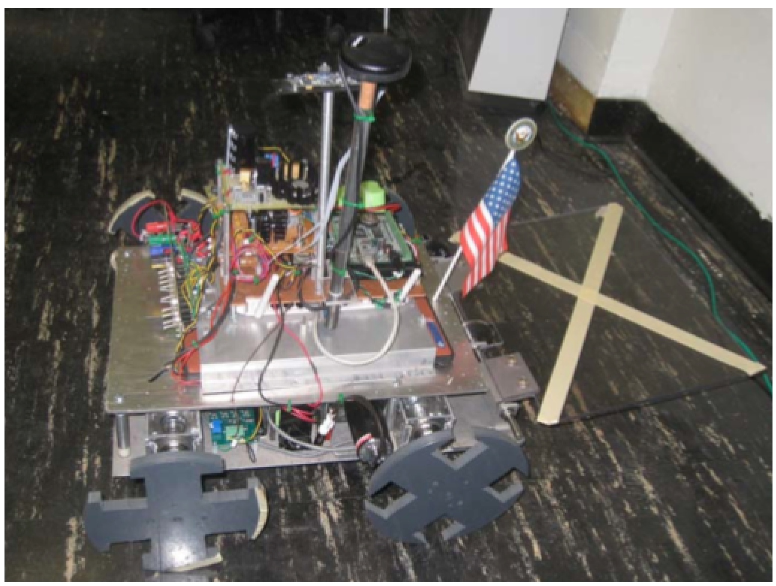

Figure 1.3: From [3], a picture of Robster

was 2/3 the length of the entire robot. The torque of the motor and gearing allowed the tail to lift the entire rear of the robot. Dynamic tests were conducted to determine the effects of using the tail for climbing assistance. These tests did not create a high centering scenario that the design is susceptible to. Some recommended design improvements reported by Holland included the incorporation of a solid-state micro-electro-mechanical systems (MEMS) inclinometer and an improved control algorithm that reduces spurious sensor data.

Dunbar developed navigation and control for Agbot in his thesis, including an effective waypoint navigation algorithm that interfaced with a Java based graphical user interface (GUI), written by Uzun, for a robot named Bender [2]. Agbot could navigate up to 10 waypoints, and report current GPS position and heading. Williamson wrote a proportional, integral, derivative (PID) control scheme to control the motors and calibrated the appropriate gains for an autonomous ground vehicle named Bigfoot [7].

1.1.2 Related Works

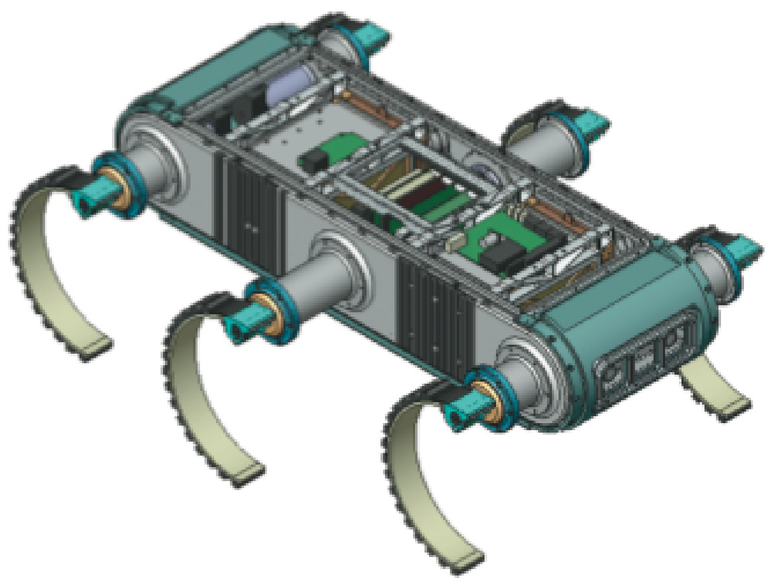

Figure 1.4: From [4], a 3D rendering of AQUA outfitted with walking legs

There are a variety of research groups that have studied platforms designed for autonomy and some use specialized methods for mobility. Select robots have also been designed for amphibious operations. AQUA is an advanced robot that has two modes of operation, the land based version in Figure 1.4 and a waterborne variant that uses flippers instead of legs [4]. The vehicle's means of locomotion are unique and together attempt to overcome the difficult transition from water to land. The Surf Zone Crawler Group from the Naval Surface Warfare Center has also focused on this operating environment in order to provide mine detection and classification [8]. Their group proposed that different unmanned systems work as a team to perform separate tasks. Another design seen in Figure 1.5, called AmphiRobot, uses a unique tail and propeller system to provide mobility in water and land [5]. This robot features a variety of sensors and servomotors that allow for object avoidance and additional autonomy. Commercial teams have also developed crawling robots that can be used to conduct non-destructive testing inside oil and chemical tanks [9]. In order to deliver accurate inspection results these adverse constraints have lead to interesting designs that require advanced sensors and control systems.

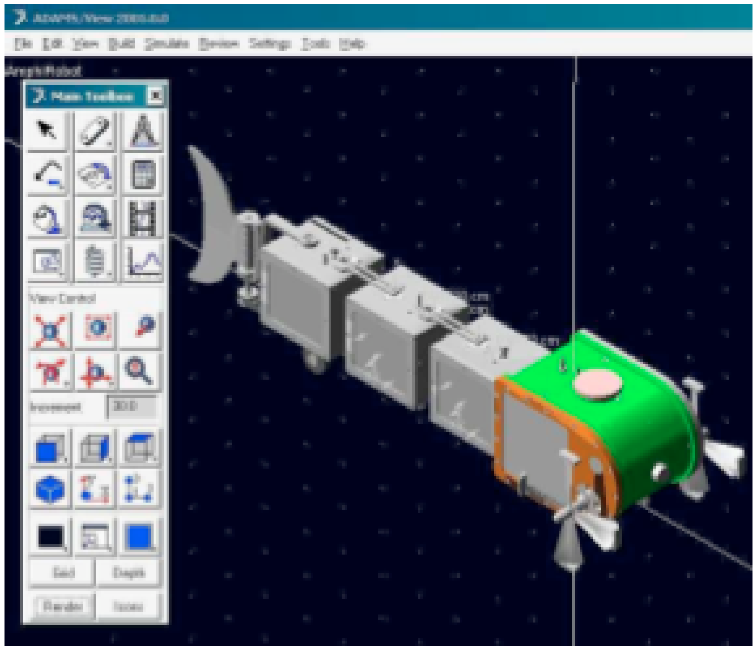

Figure 1.5: From [5], a 3D rendering of AmphiRobot

Controls and robotic operating systems are active areas of research in robotics. Autonomously integrating sensor data to navigate and perform tasks in an unmanned vehicle is a big ticket item for the Defense Department and the private sector. Numerous institutions and agencies, such as the Defense Advanced Research Projects Agency (DARPA) and (AUVSI) have sponsored many competitions and research projects in the subject. One example is the DARPA Urban Challenge, the last of which was held in 2007. Teams had to convert automobiles to negotiate a complex course in an urban environment autonomously with no human assistance [10].

The operating systems and architectural structures for controlling robots are many and varied. Previous work was done using DynamicC©. Developed for use with Rabbit Microprocessors, Dynamic C©provides multitasking capability for a robotic project. Lopez, Bigfoot, and Agbot programs were all written in Dynamic C. The Robotic Operating System (ROS) is another example of an object oriented operating system, which is open source and maintained by Willow Garage. Programs like ROS allow for rapid prototyping and integration of software. Examples include the AsTec Quadrotor unmanned aerial vehicle, Clearpath Kingfisher sea-based system, and the iRobot Roomba [11, 12].

1.2 Concept of Operations

An autonomous amphibious vehicle that could operate in the surf-zone would provide a valuable capability to the military. An inexpensive robotic platform could be deployed covertly from the sea and make its way onto the shore and replace the need to send in a human. This could be accomplished on the surface, or subsurface depending on the system.

The transition from the sea to shore is the unique aspect of this concept. Crashing waves, rock formations, and other features provide quite the challenge in negotiating its way to shore. A surf-zone robot would need to utilize multiple systems to successfully make this transition.

Once ashore, the platform can perform reconnaissance, disable mines, or deploy devices depending on the mission requirements. This is only a short list of the capabilities provided by this kind of vehicle. The sensor and mission packages could be modular to provide flexibility. Multiple sensor inputs, including GPS, inertial navigation systems (INS) and stereovision, provide positional and path-finding capabilities.

Communication both at sea and ashore will need to be handled effectively. Wireless or laser point-to-point communications will work well for the surface. Submerged navigation and communications can be handled via acoustic beacons like Seaweb [13]. The concept can be taken further where a "mother ship" style deployment system can be implemented. An offshore platform can deploy and serve as the communications hub for the surf-zone robots. Once the mission has been accomplished the robots can transition from shore to sea for scuttling or recovery.

CHAPTER 2:
Concepts for Mobility and Navigation

2.1 Mobility

2.1.1 The Use of Whegs^{TM}

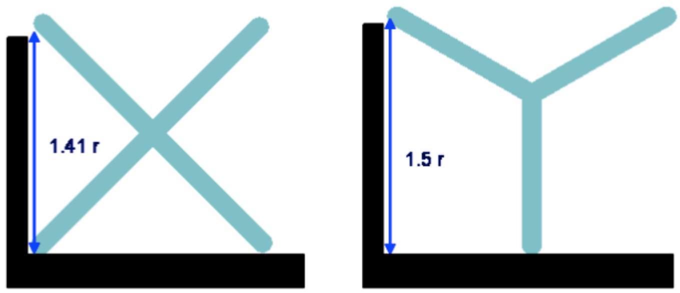

Figure 2.1: From [1], two graphic representations of Wheg^{TM}designs where left is four spoke variant and right is three spoke variant

The Wheg^{TM} has been an instrumental aspect in the design of a surf-zone robot. It allows for fast travel over smooth terrain and offers the ability to climb over obstacles. Figure 2.1 shows a basic diagram of two different variations of Wheg^{TM} design. The three spoke variant was originally introduced on a robot with six Whegs^{TM}. Their rotation was phased in a manner to always have three Whegs^{TM} in contact with the ground. This provided sufficient stability during locomotion and while standing still. The three spoke variant is also able to climb a higher step size when compared to a four spoke variant. The one disadvantage to having three spokes is that the center of rotation has a large vertical variation as the Wheg^{TM} rotates and advances the position of the Wheg^{TM}. This produces significant vibrations as the robot moves along a path. Having groups of Whegs^{TM} phased together reduces the overall undulation of the robot.

The four spoke variant helps limit the vibration and vertical variation seen at the center of the Wheg^{TM}. The step height geometry in Figure 2.1 corresponds to a 30% vertical variation for four spokes vice 50% for three spokes. More spokes offer more stability since there is one additional leg to provide support through one rotation. Ideally a complete wheel would be used to limit the vibration, however that would sacrifice the ability to climb obstacles and travel in complex terrain such as loose sand. The four spoke variation does limit the step size that it is able to climb over; however it is only a six percent reduction. Overall the Wheg^{TM} has proven successful for surf-zone operations and different variants can be used for specific applications [1].

2.1.2 Optimizing the Center of Mass

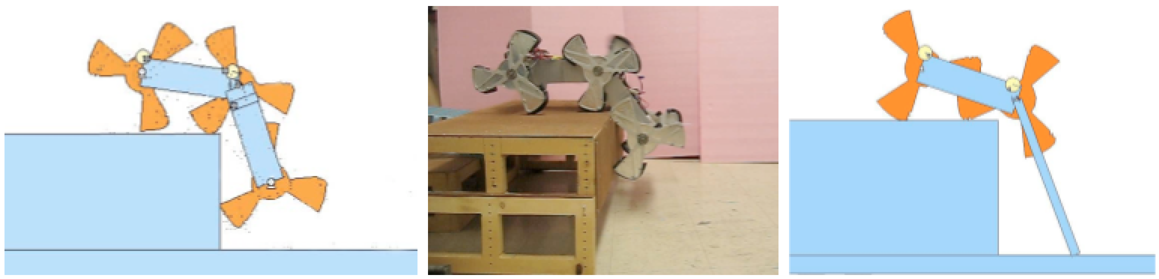

Figure 2.2: From [2]: Left image shows a previous generation robot successful climbing an obstacle in modeling environment, Center picture shows accomplishing the task by a prototype robot, Right image shows a theoretical design with a tail capable of climbing a higher obstacle

Previous generations of Wheg™ robots used an overall symmetric design consisting of a front and rear segment. These designs, one of which is shown in Figure 2.2, have had significant success at climbing large obstacles. Replacing the rear segment of the articulating body with a tail shifts the center of mass forward on the robot allowing it to climb 20 precent larger obstacles [2]. Video from various trials was carefully reviewed and it was determined that the location of the center of mass relative to the position of the center Wheg™ was the deciding factor for success. The tail acts as a support point behind the main body that can provide leverage to lift the rear of robot. The incorporation of the tail optimizes the center of mass while climbing adverse terrain and increases the overall mobility of a surf-zone robot. Further rigid body analysis and modeling should be investigated to determine more quantitative insight into this high center scenario.

2.2 Navigation

2.2.1 Waypoint Navigation

Path planning using GPS waypoints is a simple but effective means of navigating a robot through its environment. In this mode, the robot will find the heading to the destination and drive towards it. Since the destination is set by the user, autonomy in this mode is limited to driving the plant to arrive at the destination.

For this method of implementation, obstacle avoidance is possible only through the user's selection of a safe path. Implementation of object avoidance will require additional sensor input. Additional sensors can be used to implement more sophisticated navigational methods such

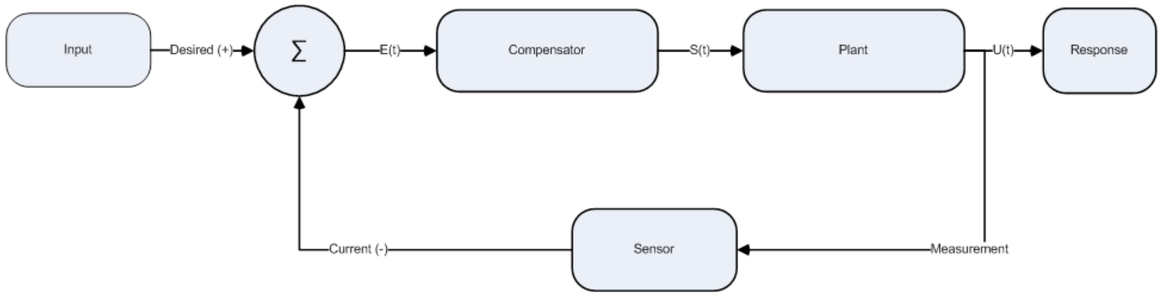

Figure 2.3: Example of PID control loop

as "bug" algorithms (that skirt around the edges of encountered obstacles) and potential field navigation [14]. Advanced path planning techniques will be investigated in future works.

2.2.2 Plant Control

A robot using a compass and GPS waypoints needs to have a feedback control to navigate successfully to its destination. Proportional, Integral, and Derivative feedback control (PID) is a popular method. PID control provides effective system response capabilities and tuning of system parameters [15, 16].

Figure 2.3 illustrates the function of the PID feedback loop that can be used with heading control. First, the desired heading is differenced with the current heading and the error signal is produced. This error, denoted $e(t)$, is then fed into the PID compensator which consists of a series of gains denoted K_p, K_i, and K_d, such that the control signal is given by:

$$u(t) = K_p\, e(t) + K_I\, \frac{1}{T_I} \int_0^t e(\tau) d\tau \; + K_D\, T_D \frac{de(t)}{dt} \tag{2.1}$$

Equation 2.1 shows the time response of a typical PID controller. The proportional gain, K_p, is a gain that is applied to the error to provide a signal. This gives the kick to the plant to get it traveling towards the destination, but may lead to "hunting." Hunting is where the robot's heading oscillates about the desired path. This typically happens when the gain is too high and is known as under-damping. If K_p is too low the response will take a long time to reach the ordered heading (i.e. over-damped), and could never reach the waypoint depending on the positional geometry.

Integral control, K_i, helps solve some of these issues. The integrator sums the error over a given period of time, T_I, and then applies the resulting gain to the plant signal. This allows the feedback loop to correct for encountered friction and inertia that produces a response offset. The error is time-averaged, and will give a boost to the signal if the outcome is slow to respond.

The final component is the derivative control, K_d, which helps improve dynamic response. This method monitors the rate of change in the error, over time T_D, which can help minimize overshoot of the desired outcome.

The signal generated by the PID control is then fed into the plant which produces a response. The feedback (current heading in this case) is then fed back into the loop and the process starts again. All, or some, of the components can be implemented depending on the desired response required.

2.2.3 Gain Scheduling

When implementing PID control, selecting the appropriate gains is vital. This does not mean K_p, K_i, and K_d need to be constant. This is commonly referred to as gain scheduling. One implementation is to change K_p, K_i, and K_d based off of the current terrain the robot is traversing. Another use is to limit the maximum gain until a threshold is reached. For example, a robot could switch off PID control if the heading error was greater than 90 degrees. If outside this tolerance, the robot would simply turn at the maximum rate until the error dropped below the appropriate level. [16]

A more detailed discussion of PID control can be found in the thesis written by Dunbar [1].

CHAPTER 3:
Design

The first step in creating Mobility Over Non-trivial Terrain (MONTe) is to dissect the concept of operations and determine characteristics that will be inherent to our design. For example, an amphibious robot could be designed to crawl on the bottom of a body of water, float at the surface, or possibly be engineered to do both. These preliminary design constraints establish core capabilities that act as a foundation for the overall design. Our team placed three constraints on the MONTe. First, the robot is watertight, vice free flood, and is positively buoyant. Designing the robot to float on the surface improves the reliability of communications, allows for the reception of GPS information, and simplifies the initial testing environment. Secondly, Whegs™ provide MONTe's land based locomotion and a tail assists in climbing terrain. Lastly, the robot is semi-autonomous vice tethered. These constraints may be altered for later builds as MONTe matures through testing and improvements.

The following provides a quick overview of the design, referencing Figures 3.1 and 3.2. For details on the mechanical design, refer to Slatt [17].

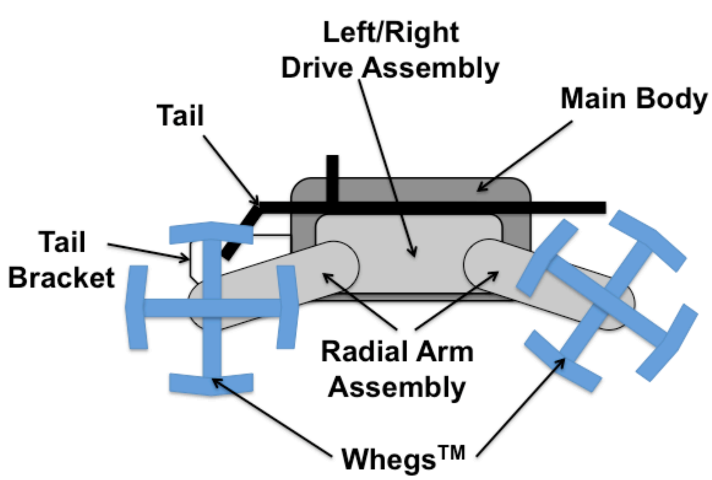

Figure 3.1: Side view of MONTe's components

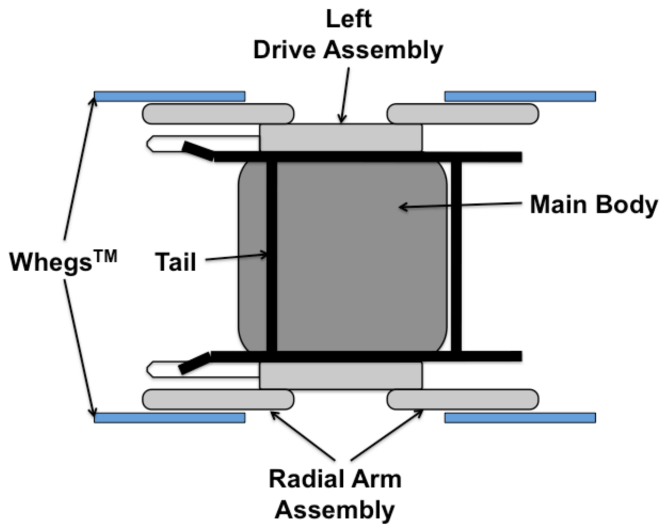

Figure 3.2: Top view of MONTe's components

3.1 Mechanical Components

3.1.1 Main Body

The challenge of making MONTe watertight led to the selection of a Pelican Case for the main body. This allows for the main access to be sealed with a gasket while allowing frequent opening and closing throughout testing. As additional penetrations are made through the case, they are sealed individually using gaskets or techniques.

3.1.2 Drive Assembly

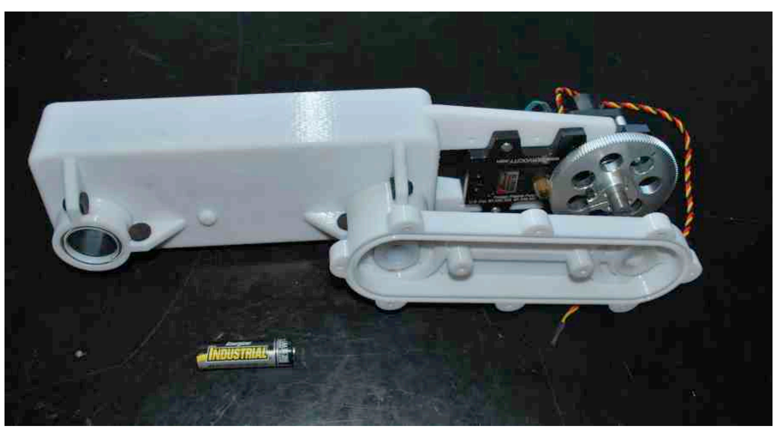

Figure 3.3: Picture of MONTe's drive assembly attached to half of a radial arm

Attached to both sides of the main body are the left and right drive assemblies. These drive assemblies contain the drive motor shafts that connect to a main drive belt. It also houses a suspension system for both the front and rear Whegs™. Limit switches are triggered if any force displaces this suspension from its equilibrium position, which is used to detect a free rotating Wheg™. The drive assembly can be seen in Figure 3.3.

3.1.3 Radial Arm Assembly

The radial arm assemblies connect each Wheg™ to its associated drive assembly. Each assembly contains a belt and pulley configuration that transfers torque from the drive assembly to each Wheg™. It comprises two halves that are joined together. One of these halves can be seen in Figure 3.3.

3.1.4 Wheg™

Figure 3.4: Picture of MONTe's initial Wheg™design

This Wheg™ design, Figure 3.4, is a four spoke variant. As discussed in Section 2.1.1, it offers less vibration over smooth terrain, but limits the step size for which the Wheg™ will climb. There has also been a decrease in the total number of Whegs™from six to four. The Whegs™ are designed to be easily removed and allows for quick implementation of changes to the Wheg™ design. The shape and symmetry of the Wheg™ affect its ability to climb and will continue to be adapted throughout the development process.

3.1.5 Water Jets

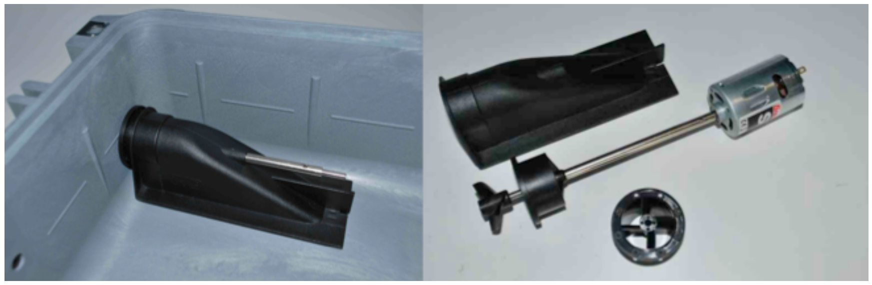

Figure 3.5: Picture of MONTe's future water jet, left image shows the placement and right image shows the components

Water-borne locomotion will be provided with a water jet system, as shown in Figure 3.5. The motors, ducting, and impeller will be housed in the main body. The ducting acts as a new pressure boundary within the main body. This allows the jet motors to be isolated from the water while providing thrust.

3.1.6 Tail

One of the major advancements of MONTe is the incorporation of an autonomous tail. The tail was introduced to the overall design in order to overcome a susceptibility to high centering by moving the center of mass forward. The tail is designed to assist in climbing obstacles and to self-right the robot in the event it becomes flipped over. Tail operation is modeled for to determine performance, which will be discussed in the Section 4.3.

Mechanical Design of Tail

The tail for MONTe is relatively simple in construction and provides the mechanics for an initial proof of concept. The most apparent simple design is a rigid flap that attaches to the robot at a joint as seen conceptually in Figure 1.2. Two independent joints provide rotational motion along the same axis. To prevent the tail from obscuring any sensors, a "wire frame" structure is used, Figure 3.6. Additionally, it reduces the overall weight of the tail and maintains a similar level of strength.

Steel tube, 3/8" outer diameter with a 1/16" wall thickness provides the main structural support to create the tail. When in a stowed position, Figure 3.7, the tail wraps around the main body case of MONTe, as in Figure 3.6. The frame is supported by a cross member that bends over the top of the case. The cross member is placed in an optimal location. It is positioned close to the control joints to provide support while ensuring enough distance to prevent interference with components when in the extended position. The tip of the tail is left hollow to allow extensions

Figure 3.6: A picture of MONTe with the tail in the stowed position

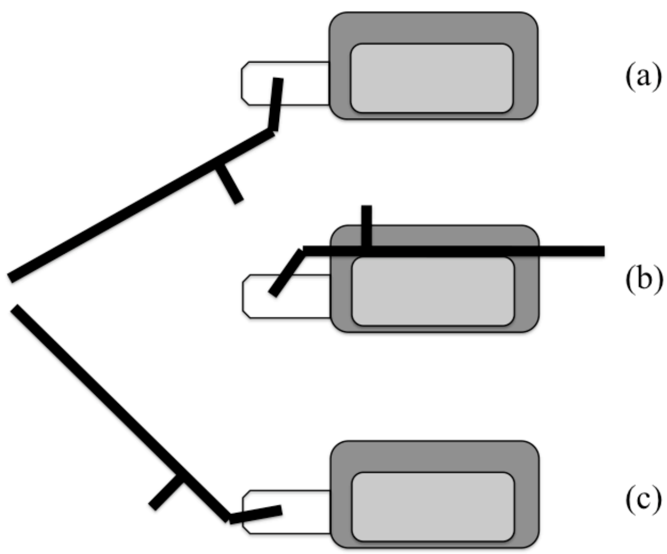

Figure 3.7: MONTe's tail positions (a) Down (b) Stowed (c) Neutral

to be inserted into the tubes. These extensions allow for changing the overall length of the tail as necessary throughout design and testing.

The tail is 11.8 ounces and is attached to a high torque servo drive mechanism. These servo components are attached to the drive assemblies using steel brackets and fasteners. The combined bracket and servo drive mechanism weighs 16.2 ounces. Later designs will reduce the overall weight by replacing the steel bracket with lighter weight polycarbonate materials. Combining the tail bracket with the drive assembly will eliminate fasteners and be a necessary design improvement to maintain watertight integrity of the overall design. This single unit will encase the servo drive gears shielding it from debris and sand and thus preventing damage to the gears.

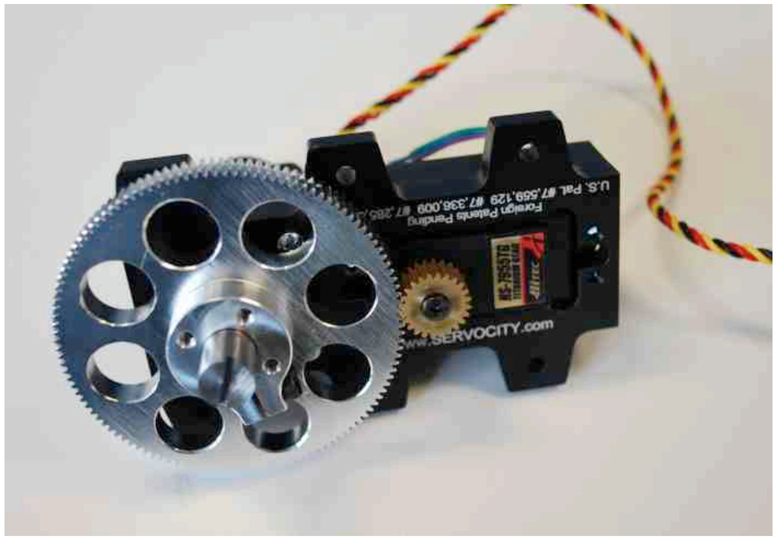

Figure 3.8: A picture of the high torque servo drive mechanism

Tail Drive Mechanism Design

The tail drive mechanism consists of a titanium geared hobby servo, HS-7955TG, and a supplemental gearbox, Figure 3.8. The stock servos were modified to allow continuous rotation of the servo and incorporate a new potentiometer into the gear train. These high torque servos provide 95–3300 oz-in of torque [18].

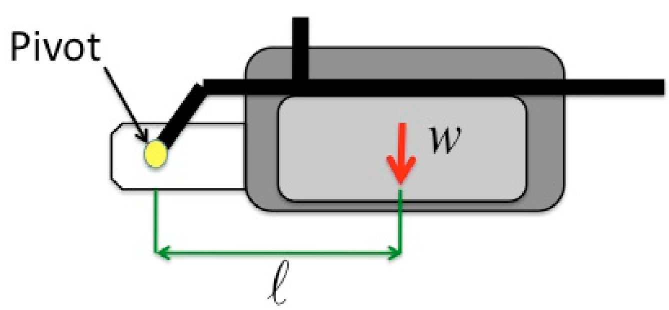

Figure 3.9: A graphic showing a simple lever arm that can be used to estimate the required torque output of the servo drive mechanism

$$\mathbf{Torque} = \mathbf{r} \times \mathbf{F} \quad \Rightarrow \quad |\mathbf{Torque}| = |\mathbf{r}||\mathbf{F}| \sin\theta = lw \tag{3.1}$$

Preliminary calculations were necessary to provide an estimate of the torque required to operate the tail. The general torque expression is given by Equation 3.1, where F is the applied force vector and r is the displacement vector from the joint to the applied force. Based on a weight estimate of the entire robot and the location of the tail joint, that equation reduces to a simple expression. Figure 3.9 depicts the moment arm that transmits the torque of the center of mass on the tail joint. Based on an anticipated final weight (w) of 20 lbs. and a moment arm (l) of 8 in., the estimated total torque would be 2600 oz-in. Since two servo mechanisms will be used to drive the tail, each unit would need to supply roughly 1300 oz-in, which is governed by the following equations [19]:

$$T_M = K_t \phi I_a \tag{3.2}$$

$$T = J_{eq}\ddot{\theta} + F_{eq}\dot{\theta} \tag{3.3}$$

$$J_{eq} = J_a + \frac{1}{N^2} J_L \tag{3.4}$$

$$F_{eq} = F_a + \frac{1}{N^2} F_L \tag{3.5}$$

As seen in Equation 3.2, torque of the electric tail motor is proportional to the current that it draws, I_a, flux for each pole, ϕ, and a constant, K_t, related to the physical design of the motor. This is a design issue for MONTe since current is provided from a limited battery system, see Section 3.4. Even if ample current is available, electric motors will stall if the combined friction and inertia applied to the motor are too great. The mechanical equations of motion for the joint are given by Equation 3.3. This shows that torque is proportional to the inertia, J_{eq}, and friction, F_{eq}, through either the angular acceleration, $\ddot{\theta}$, or angular rate, $\dot{\theta}$. When considering a joint driven by an electric motor, the inertia and friction can be divided into two components, the armature of the motor, a, and the external load, L, of Equations 3.4 and 3.5. When considering MONTe's self-righting high torque scenario, the load produces an overwhelming effect over the armature. By applying a mechanical gear, the torque is transmitted from a longer moment arm and acts to reduce the inertial and frictional effect on the motor. Equations 3.4 and 3.5 also show that by selecting the proper gear ratio, N, the inertial and load can be dominated by the armature and prevent the motor from stalling. The tradeoff is that the tail rotation speed will be reduced. Testing in Section 4.3.1 further investigates the torque required for MONTe's servo drive mechanism.

3.2 Operating System Architecture

Robots designed for reconnaissance must be capable of several tasks if they are to be useful to the organization that operates them. A robot must first be able to get to the target location. Once there, it must be able to orient itself in its environment and the region of interest. Its sensor package must be able to find and record data of interest. Finally, the robot must be able to communicate at some point with the operator in order to complete its mission.

The level of autonomy becomes a vital part the design implementation. At one end of the spectrum is an unmanned vehicle that is fully user-operated. While easier to design in a technical sense, this can be prohibitive in the man-hours required to operate it. Furthermore, the user will most likely be only able to operate one unmanned system at a time. The other end of the spectrum is a system designed to require no user input beyond mission parameters.

MONTe is designed to operate in the middle of the spectrum. To be semi-autonomous in nature, MONTe can communicate, navigate and be controlled by the operator as the situation warrants.

The primary operating system for MONTe is located on the LPC-100 computer. The main program's responsibilities include communications, navigation, plant control and eventually stereovision. The architecture of the operating system is based off the Robot Operating System (ROS).

3.2.1 ROS Overview

ROS is an open-source, meta-operating system designed for robotic applications. It uses a peer-to-peer network messaging system between different processes. The different processes are loosely coupled by being compartmentalized. It is not a real time operating system. ROS is also designed to allow for portability between different robotic projects [11].

Fundamentally, ROS operates as an object-oriented messaging service. This allows communications between different processes. These processes are called nodes in ROS, which are nothing more than software code and drivers. This provides a great framework for incorporating sensors and other devices into the robot. Incorporating a laser rangefinder involves writing a driver to interface with the device and then transmit the information to other nodes that need the information.

The first mode of communications is a traditional service style messaging system. One node waits for a signal from another node before transmitting a response. This provider/client system is effective but requires the nodes to be more coupled than is desired for this project.

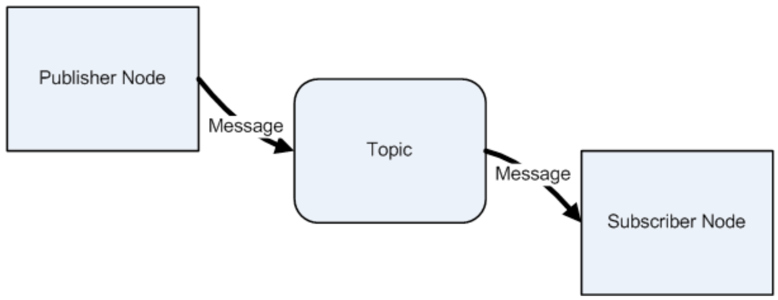

Figure 3.10: Illustration of a basic ROS publisher/subscriber interaction.

MONTe uses a publisher/subscriber framework for handling inter-nodal communications. The framework is made up of publisher nodes, subscriber nodes, and topics. Figure 3.10 provides a basic illustration of the interaction of nodes through a topic. The publisher generates a message (described by a user-defined `.msg` template) that is then published to a topic. The subscriber will read from the topic, via another message, by a polling process called "spinning". The publisher and subscriber are decoupled because neither directly sees the other. Taking the publisher/subscriber concept further, multiple nodes can subscribe or publish to the same topic. A node has the capability to both publish and subscribe. This allows for complex interactions between nodes, and the decoupling makes debugging nodes easier [20].

3.2.2 Functional Architecture

MONTe's program has several design goals for this version. Conceptually, the architecture needs to provide a foundation for current and future work. Physically, it needs to interface with sensors, control the motors, and communicate with the operator. Behaviorally, it needs to fuse the sensor data and user inputs to navigate effectively. Finally, the structure of the program needs to be de-conflicted so that the nodes are not interfering.

Figure 3.11 outlines the basic structure of MONTe's main program. It illustrates all nodes, topics, and hardware interfaces. It further illustrates the flow of information and commands through the network of nodes. A more detailed discussion of the individual nodes and topics will follow.

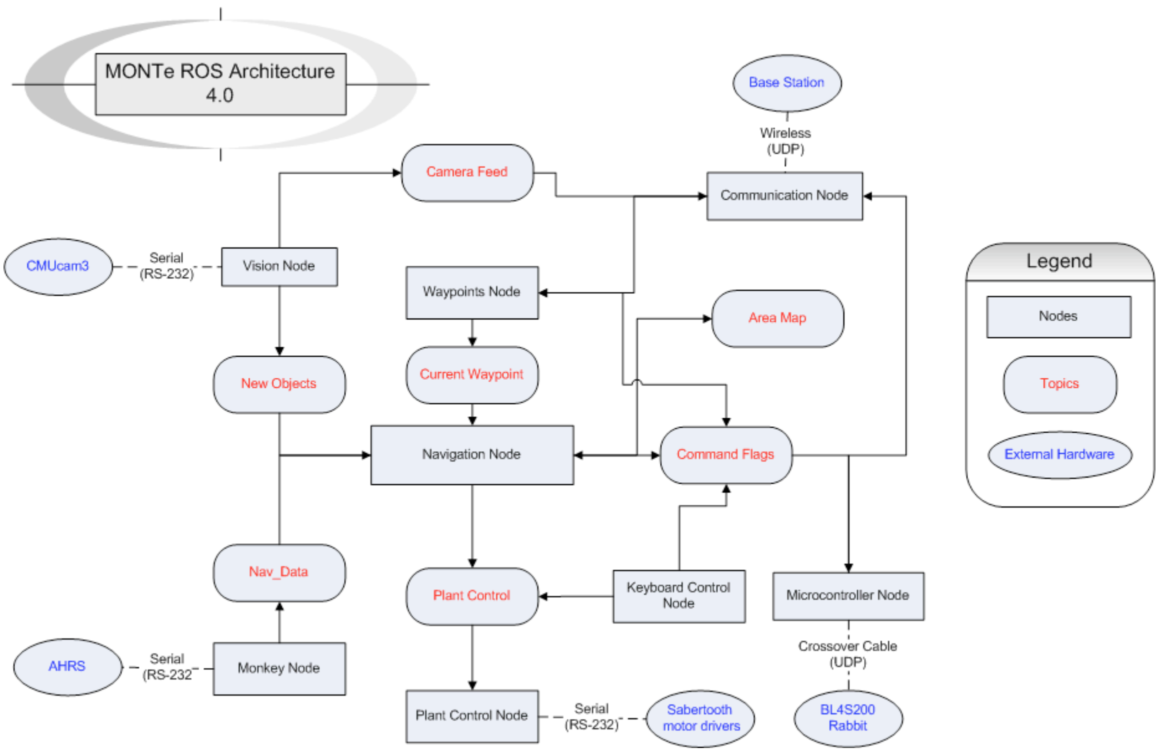

Figure 3.11: Diagram of MONTe ROS Architecture

ROS Nodes

Nodes are vital to how ROS operates. Fundamentally, nodes in ROS are functions that perform tasks that an autonomous system needs to accomplish. Messages and topics allow each node to operate independently, as each only sees the topics it is subscribed to.

Figure 3.12 shows typical initialization commands used in a ROS node. Upon launch, each node will invoke `ros::init` and `ros::NodeHandle` to initialize the node and provide a name or "handle" for `ROScore` to interact with. Next, all publishers and subscribers are set up via the advertise/subscribe functions. Finally, the messages needed to talk to the topics are initialized. In this case, `MONTe::Plant_Command cmd` initializes a message handle `cmd` of message type `Plant_Command.msg`. This would enable messages to be sent to the ROS topic `Plant_Command_T`.

At this point the node will enter a loop to perform calculations. The first or last item in the loop should be to "spin" ROS in order to poll all topics. `ros::spinOnce()` polls any callback functions that have been declared in the node. The node can also publish at any point of the loop using the `.publish()` command.

```
/*  Sample code to demonstrate ROS concepts     */

int main (void);
{
        ros::init(argc, argv, "MONTe_Navigation");  // Set up ROS node

        ros::NodeHandle n;  // Set up handle for this node

        // Set up all publishers for node
        ros::Publisher plt_cmd_pub = n.advertise<MONTe::Plant_Command>("Plant_Command_T", 1);
        // Set up all subscrivers for node
        ros::Subscriber sub_cf = n.subscribe("Command_Flags_T", 1, Command_FlagsCallback);
        // Set up message handles
        MONTe::Plant_Command cmd;
        MONTe::Command_Flags flags;
        /* More code here */
        while(ros::ok())
        {
                ros::spinOnce();        // Poll topics
                /* Perform calculations */
                plt_cmd_pub.publish(cmd);  // Publish data to topic
        }
}

void Command_FlagsCallback(const MONTe::Command_FlagsConstPtr& flags)
{
        Cmd_Flags.autonav = flags-> auto_nav;
        Cmd_Flags.mode = flags-> nav_mode;
        Cmd_Flags.route = flags-> incoming_route;
} // end callback
```

Figure 3.12: Sample code illustrating ROS concepts

MONTe's nodes can be conceptually divided into two types: demand and continuous. The demand nodes consist of the `Waypoint Control` and `Keyboard Control`. These do not run in a continuous loop, instead wait for input prior to executing. The other nodes are designed to be running in a continuous loop. These nodes will run processes, and poll and publish to topic at a rate of 4 Hz. The nodes do this by taking advantage of the ROS `loop_ratesleep()` function that enforces the desired frequency. The system cycle rate of 4Hz was selected based on updating the navigation algorithm at a sufficient rate, but is open to further optimization.

Communications Node

This node covers the communication with the base station and operator. The node communicates via UDP protocol over a series of ports to keep data streams separate. The communication types are divided into a series of channels. Each channel, upon receipt or transmission, will publish data and control flags to various topics. An example would be manual control. Upon receipt of a manual command, the `Communications` node would process the input and publish the pertinent data to ROS topics like `Plant_Control topic` and `Command_Flags`. The other communication channels include `Waypoints` (receive), `Navigation` (send), and

Message (send).

Waypoints Node

The Waypoint node processes operator generated navigation data. The waypoints are stored in the node using dynamically allocated memory, and are arbitrarily limited to ten waypoints. The stack has functionality to switch between waypoints, create more and to delete the stack.

Waypoints node subscribes to the New_Waypoint and Command_Flags topics to manipulate and input new waypoints into the stack. The node publishes the current destination to the Current_Waypoints topic.

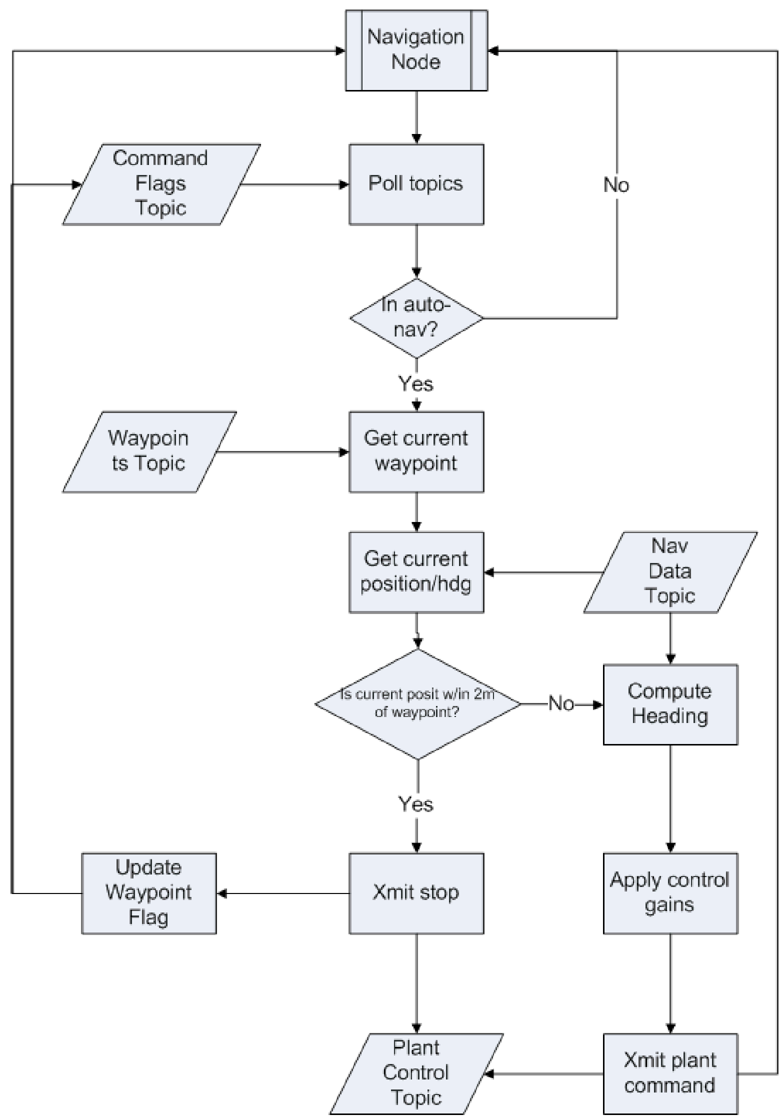

Figure 3.13: Flowchart for MONTe's Navigation Node

Navigation Node

The `Navigation` node is the primary behavior generator for MONTe. In Figure 3.11, `Navigation` resides at the center to represent its importance in the overall architecture. All major data paths begin or end with the navigation node.

The current program will allow MONTe to travel to a desired waypoint without object avoidance. Figure 3.13 details the operation of the navigation routine. `Navigation` polls `Command_Flags` and `Current_Waypoints` to verify MONTe is in auto-nav mode while updating the next destination. Positional data (compass and GPS) is received from the `Nav_Data` topic. Current position and the current waypoint are compared and appropriate plant control data is published to `Plant_Control` topic. When the current destination is reached a flag is updated on `Command_Flags` so that the `Waypoint` node can send the next waypoint.

Monkey Node

The `Monkey` node is the driver associated with the Monkey Attitude Heading Reference System (AHRS) unit. In the current implementation, the node is a publisher only. Its primary function is to receive GPS, compass, and velocity data for publishing to the `Nav_Data` topic. In the future, its secondary function is to allow for manual control of the tail or adjust the autonomous functions of the tail.

Plant Control Node

The `Plant Control` node is simple in operation. It is the driver for interfacing with the Sabertooth2x12 motor drivers. It handles this by polling the `Plant_Control` topic to receive commands. They are then parsed and transmitted via RS-232 serial port to the motor drivers.

Keyboard Control Node

`Keyboard Control` allows the user to control MONTe manually via a virtual network client (VNC) server. Control is rudimentary with forward, reverse, stop, left, and right turns possible. It also allows the speeds and turning rates to be adjusted during operation. Upon receipt of a command, the node will publish the motor commands to the `Plant_Control` topic, while updating the `Command_Flags` topic to manual control. This will take MONTe out of autonomous navigation.

Waypoint Control Node

`Waypoint Control` allows the user to input and delete waypoints, and then execute the route remotely via VNC. The maximum number of waypoints is set to ten. Each waypoint is

```
# Plant_Command.msg
#
# Basic message for manually controlling MONTe in simplified serial mode.

# Speed command for left motor.  Range is 1(Full Reverse)-> 64 (Stop) <- 127 (Full Forward)
uint8 left

# Speed command for left motor.  Range is 128(Full Reverse)-> 192 (Stop) <- 255 (Full Forward)
uint8 right
```

Figure 3.14: Example of a ROS message

set up with latitude, longitude (in decimal degrees), waypoint number, and an action. Actions are set up for future use for more sophisticated control. Once a route is generated, the user will send the route which feeds into the `New_Waypoint` topic. `Command_Flags` topic will also be updated to autonomous navigation which will take MONTe out of manual control.

ROS Topics

Topics allow ROS to transmit data between nodes. The nodes only see the topics they publish and subscribe to, which decouples them. Each topic is communicated to and from via a message. Valid data types include integers, floating point, characters, and strings. A sample message is shown in Figure 3.14. This simple message holds to unsigned, 8-bit integers, and can be used to send to or receive from a topic.

The main type of topics that MONTe utilizes transfer sensor data between the nodes. For example, `Nav_Data` provides the current position and heading of MONTe to any node that requires that data. Other messages transfer command data to topics such as `New_Waypoint` and `Current_Waypoint`. The messages in this case send waypoints with any additional information required for processing the data.

`Command_Flags` topic is a special topic that stores all behavioral flags that control MONTe's operation. The current flags it stores are `auto_nav`, `nav_mode`, and `incoming_route`. These flags allow MONTe to switch between manual and autonomous control modes, indicate when a waypoint is reached, and to warn when a new waypoint route is in the queue.

3.3 Control System Hardware

The control system hardware is housed in the main body of MONTe, shown in Figure 3.15. Individual components are mounted onto a power module that acts to support the electronics and route wiring for the devices. This assembly also organizes the switches that activate each power bus and device.

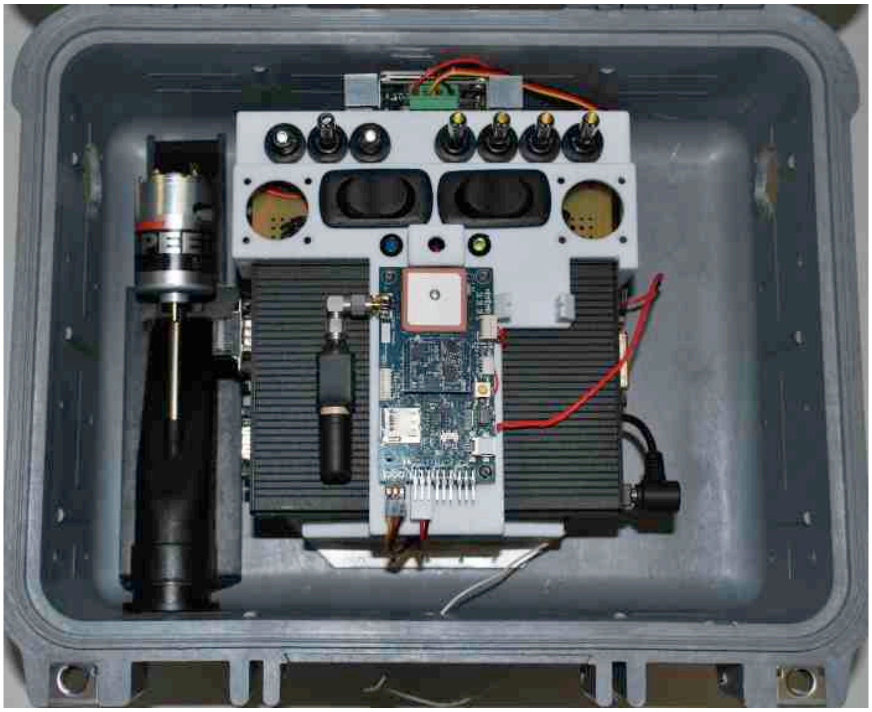

Figure 3.15: Picture of the internal design and component placement of MONTe

3.3.1 Main Processor

Figure 3.16: Picture of the Stealth LPC-100

The main computational device for MONTe is a Stealth LPC-100 mini-personal computer, Figure 3.16. It was selected due to its small size. It weighs 1.2 lbs, and its dimensions are 4.0”(W) x 6.1”(D) x 1.45”(H). This small form factor is ideal for use in a self-contained robot.

The computer runs on a 1.9GHz Intel-Celeron processor with 4GB of RAM. This provides a robust capability for running the operational program, as well as serving as the communications hub. The operating system is Linux/Ubuntu 10.04 that allows a stable version of ROS to be run.

3.3.2 Monkey Board

Figure 3.17: Picture of the 2010 Monkey Board produced by Ryanmechatronics LLC

The Monkey 2010 platform, Figure 3.17, is a versatile circuit assembly that can be used to control autonomous vehicles. There are multiple capabilities inherent to the board, but some of the main functions include: a Cortex M3 (ARM 7) processor, U-Blox NEO-5 GPS module, barometric pressure sensor, I/O ports, pulse width modulation (PWM) servo outputs, and status LEDs. This board is designed to accompany a CHIMU (product name) AHRS. A robust software suite allows for easy user interface and control algorithm development. Combining these modules allows MONTe to acquire its GPS position and know its spatial orientation: roll, pitch, and yaw. The Monkey is limited in its video processing capabilities and therefore only used for advanced sensing and tail control.

3.3.3 Motor Driver

The Sabertooth2x12, shown in Figure 3.18, is a 6-24V motor driver designed for analog, radio control (RC) and serial control applications. It can operate at up to 12A continuously and control two sets of motors per channel. Serial control in simplified mode was selected for controlling MONTe. Single byte commands are sent via RS-232 protocol to individually control each set of Whegs$^{\text{TM}}$. The Sabertooth2x12 has the capability to control up to eight different channels simultaneously using packetized serial mode. Finally, the motor driver can operate in a ramping mode that provides smooth acceleration upon receipt of commands [21].

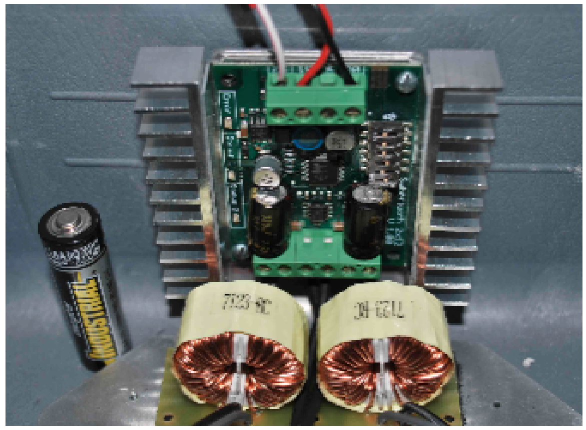

Figure 3.18: Picture of the Sabertooth 2x12 motor controller

Figure 3.19: Picture of the Belkin Wireless Adapter

3.3.4 Network Card

Wireless communications is provided by a Belkin N Wireless Adapter (F5D8053), Figure 3.19. The USB device has data rates of up to 300Mbps under USB2.0 interface. New drivers were necessary to get the device to work correctly under Ubuntu 10.04.

3.3.5 CMUcam3

The CMUcam3, Figure 3.20, is a (352x288) RGB color camera designed for open source development for a variety of applications. It is capable of performance up to 26 fps, and can process images onboard prior to downloading to another computer. The CMUcam3 will be used for future implementation of robotic stereovision for obstacle avoidance.

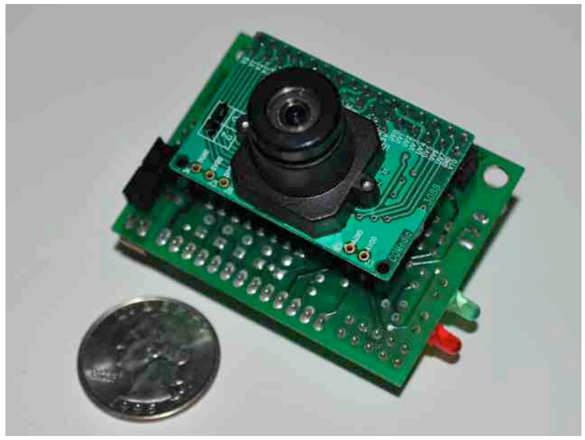

Figure 3.20: Picture of the CMU camera

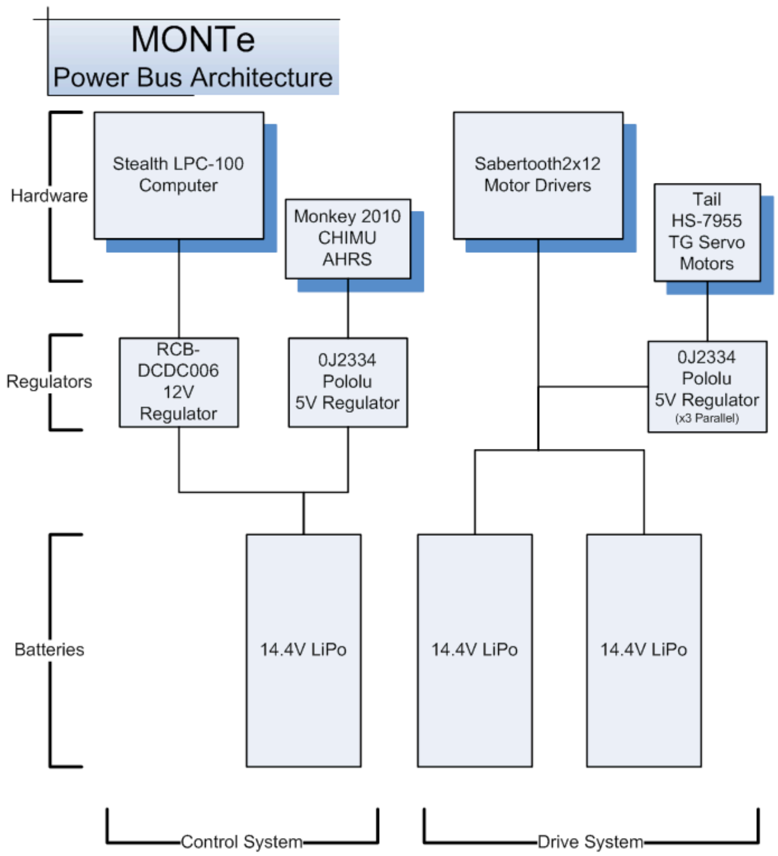

Figure 3.21: MONTe Power Bus Architecture

3.4 Power Bus

MONTe's power bus was designed to power both the control hardware (processor, sensors), and the drive hardware. Furthermore, the power bus provides the appropriate voltage regulation and protection necessary for operation of MONTe. The overall architecture is detailed in Figure 3.21. The power for the bus is provided by stacks of Lithium Polymer (LiPo) batteries. Each stack provides 14.4V to the bus. A single stack powers the Control System, while two stacks wired in parallel power the Drive System.

The Control System consists of the LPC-100 computer, and the Monkey navigational unit. The LPC-100 is fed off a RCB-DCDC006 12V Regulator. The Monkey is fed by a 0J2334 Pololu 5V Regulator. Future sensors and control hardware will be incorporated on this bus.

The Drive System consists of the Sabertooth2x12 (unregulated), and the HS-7955 TG Tail Servo Motors. The servo motors are powered by three parallel Pololu 5V Regulators. This provides the necessary current to operate the tail in self-righting mode. The two LiPo battery stacks are in parallel to provide extended operating time.

3.5 Communication Paths

Transferring data is a vital component of any autonomous or unmanned system. MONTe uses a variety of paths for external and internal communications detailed in Figure 3.22. Both current implementation (solid lines), and future communications paths (dashed lines) are illustrated. The LPC-100 serves as the central hub for both the internal communication paths (hardware), and for external communications with the base station.

3.5.1 Internal Paths

The internal communication paths for MONTe are to transfer information between the different sensors and processors. The two main formats utilized are RS-232 serial communications and User Datagram Protocol (UDP) communications.

UDP is an Internet Protocol that operates in datagram mode. This provides a quick way of sending packets of information both over hard connection (cross-over cable) or via wireless.

Motor Control

The LPC-100 communicates with the Sabertooth2x12 motor drivers via RS-232 over COM1 port. The motor drivers are operated in simplified serial mode and receive command bytes for each individual motor. The RS-232 signal must be stepped down to 0–5V TTL signal via an optical isolation circuit. The implementation is accomplished via a custom C++ library based on code from Reference [22].

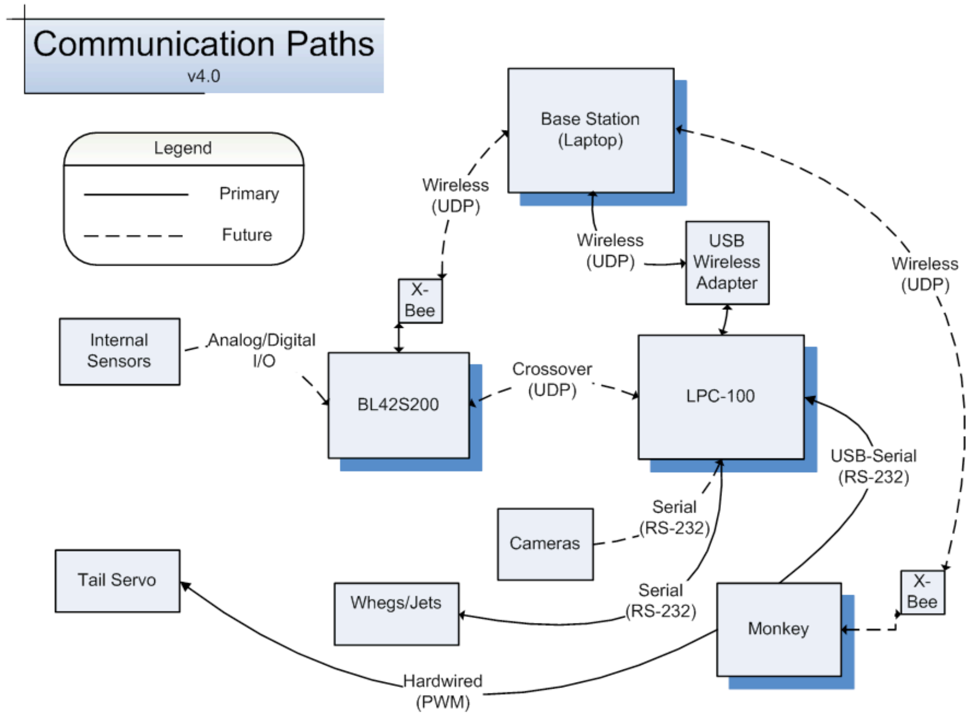

Figure 3.22: Diagram of communication paths for MONTe

Monkey

The transfer of navigation data and flags is handled by a serial-USB interface. This allows better allocation of serial communication ports on the LPC-100. GPS data is sent in NMEA format. Compass data and control flags are also sent over the path.

CMUcam3

The stereovision system will be integrated in the future via RS-232 on COM2. One camera will be the master and be connected via its UART to COM2. The slaved CMUcam3 will be connected via the second general purpose input/output port (GPIO) on the master CMUcam3.

Microcontroller

MONTe has the functionality to connect with a microcontroller via UDP using a cross-over cable. This provides a quick way of incorporating a microcontroller for integrating additional sensors. The communication protocol is designed to work with a Rabbit BL4S200 microprocessor.

VNCserver

MONTe runs a VNCserver in the background so the operator can make adjustments while active. Changes to the program and node architecture can be performed depending on needs of testing or the mission.

3.5.2 External Paths

External communications is how MONTe will interact with the operator during testing and missions. MONTe utilizes UDP IP communications with the base station through the wireless USB adapter. UDP is simple to implement and requires less processor overhead from the LPC-100. This will allow MONTe to receive commands, and transmit sensor data to and from the base station.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

4.1 Component Interfaces

4.1.1 Motor Driver

The Sabertooth2x12 dual 12A motor driver was tested in both simplified and packetized serial modes. The Sabertooth2x12 ramping function was also tested to show a smooth acceleration of the motors. Simplified serial mode was implemented in the design due to its simplicity and effectiveness. However, simplified serial was found to be incompatible with the ramping feature of the Sabertooth2x12. Simplified serial requires individual commands for each set of motors. It was found that sending the command for the second motor would override the first command if it had not reached the ordered speed.

MONTe is currently working in simplified serial with no ramping function enabled. MONTe can effectively move using this mode. The lack of ramping is of concern due to wear on the drive train. However, this is acceptable for the current iteration.

Packetized serial would solve this issue, but operation in that mode was not consistent. The first couple of commands to the plant would work as expected. However, errors would creep in to the motor driver as the values changed yielding nondeterministic behavior. Packetized serial will be necessary for implementation of the water jets, so further research and development is necessary.

4.1.2 Navigation Data

The interface between the CHIMU AHRS and the main program on the LPC-100 is done via RS-232 protocol at 115,200 Baud. Navigation data was successfully received using buffered serial at a rate of 4 Hz. The main data transferred includes: the number of satellites tracked, fix quality, three-dimensional velocity, altitude, and heading. Latitude and longitude are in decimal degrees to four places. 4Hz was selected to synchronize with the rest of the ROS program architecture to provide a constant update of heading information for the navigational control and could be further optimized in the future.

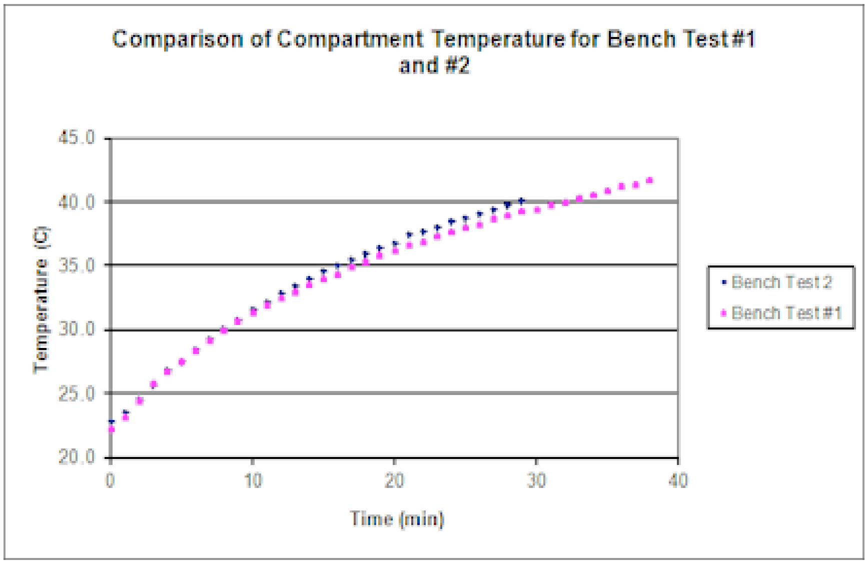

Figure 4.1: Comparison of compartment temperature profile characterizations

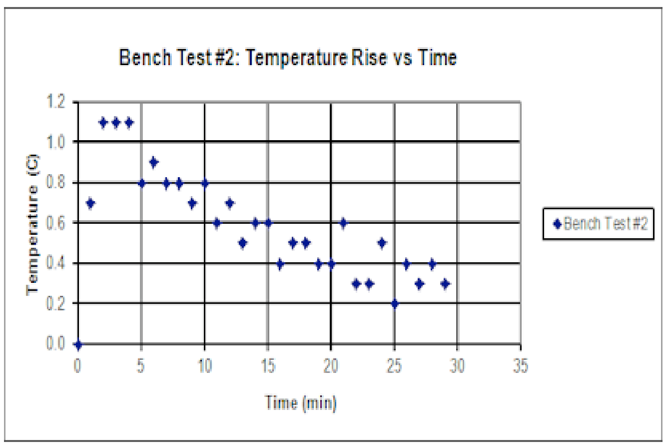

Figure 4.2: Compartment temperature change over time.

4.2 Compartment Temperature Profile

Because the LPC-100 computer has a recommended maximum operating temperature of 40°C, two bench tests were performed to determine a reasonable high-end limit for continuous operation while the compartment was sealed. The LPC-100 was started inside the chassis and employed a continuous counting program to stress the processor. The compartment temperature was monitored during operation after the compartment was sealed. The first test was done with the LPC-100 running off of external power, while the second was run off a 14.4V Lithium Polymer battery stack in the design configuration.

Figures 4.1 and 4.2 shows the compartment temperature profile generated by both bench tests. The initial internal compartment temperature for both tests was 22°C. The first test took 32 minutes to reach the critical temperature, while the second bench test took only 28 minutes on battery power. The inclusion of internal power dropped the time by 5 minutes and proved that the temperature profile is a concern. Future work will include heats sinks to manage the compartment temperature.

4.3 Mobility

4.3.1 Self-Righting

Modeling Environment

One of the major advancements of MONTe is the incorporation of an autonomous tail. To anticipate the behavior of the robot and improve the design process, MONTe was modeled using Working Model 2D (WM2D). This simulation environment models Newtonian equations of motion for interacting bodies and displays the output in an intuitive user interface [23]. It allows for interactive simulations that can receive input from user controls, scripts, and other applications, such as Excel and MATLAB. One drawback is that the software only models in two dimensions and therefore does not allow for three dimensional terrain and assumes stability in the roll direction. MONTe was modeled using approximate geometries and weights. This modeling environment provides a proving ground for various designs and control algorithms without requiring a test platform.

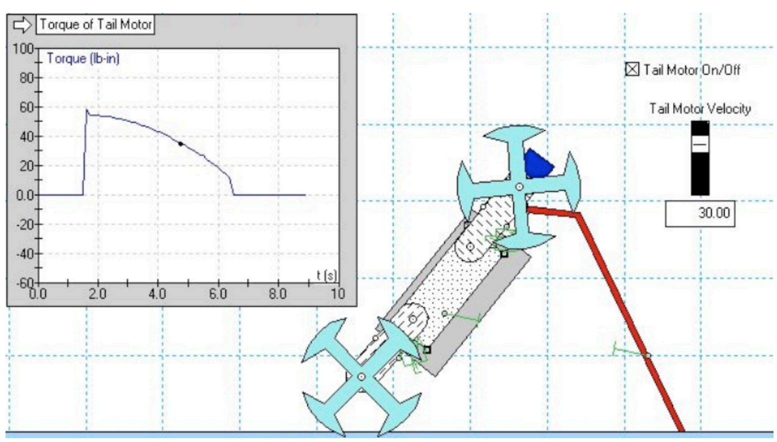

Figure 4.3: WM2D Model of MONTe rotating its tail from the neutral position to the stow position while upside down

Self-righting creates the most limiting condition for the required torque of the servo drive mechanism. MONTe was modeled early in the design phase to estimate the necessary gear ratio for

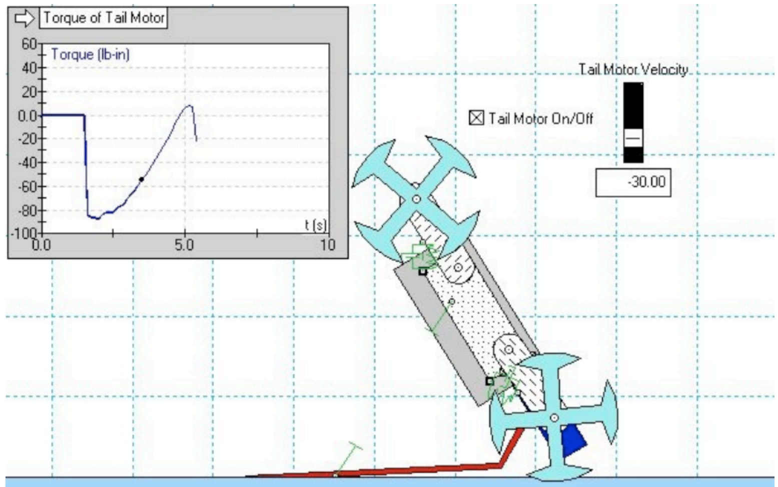

Figure 4.4: WM2D Model of MONTe rotating its tail from the stow position to the neutral position while upside down

the servo. There are two different scenarios in which the servo drive mechanism would be subjected to a large load. Figure 4.3 shows a model of MONTe righting itself from an initial neutral tail position. As the tail is retracting to the stow position, the servo drive mechanism is exerting a maximum torque of approximately 60 lb-in. This corresponds to 960 oz-in. Figure 4.4 shows a similar model except the tail is initially stowed. This is the most limiting situation and requires the highest torque from the servo drive mechanism. The maximum magnitude of torque required is approximately 85 lb-in, which corresponds to 1400 oz-in. This value closely agreed with the torque expected from our design criteria and a simple lever arm from Section 3.1.6.

The two gears selected for use in MONTe have torque ratings corresponding to 1250 oz-in or 2150 oz-in, based on supply voltage of 4.8 V [18]. Selection of a specific servo drive mechanism, with either a 5:1 or 8.6:1 gear ratio, places a constraint on the servo drive speed. The rotation speed will either be approximately 45 or 75 °/sec based on the same supply voltage. This rotation speed will dictate the time it takes for MONTe to right itself, but will also impact the time response of the tail when climbing obstacles. Since the model shows that MONTe requires at least 1400 oz-in, the 8.6:1 gear ratio will be necessary to right MONTe.

Field Trials

MONTe was subjected to tests to determine if self-righting would be possible. At the time of testing MONTe weighed 19 pounds, just below the anticipated weight of 20 pounds. Initial tests showed that MONTe intermittently succeeded to flip with a 5:1 gear system installed. The initial reason for failure was due to reaching the upper limit on current supplied to the servo drive mechanism. Additional regulators were placed in parallel to provide sufficient current. Success with the 5:1 gear ratio improved, however it was still intermittent and required upgrading to the 8.6:1 gears. Tests were then conducted with repeated success using the 8.6:1 gear ratio. Figure 4.5 shows MONTe successfully righting itself with those gears installed. This scenario is from the stowed position and places the servo drive mechanism under the largest load. Success from this position indicates that MONTe will be able to right itself from any variation of the scenario. This also ensures that there is substantial torque available to lift the rear of MONTe for climbing assist.

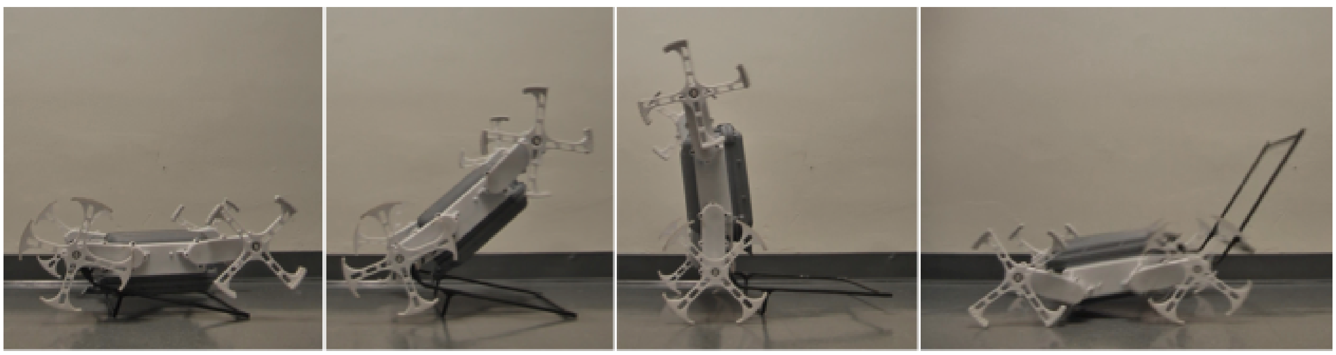

Figure 4.5: Clips of video showing MONTe righting itself from the limiting scenario

4.3.2 Climbing Assist

Modeling Environment

MONTe was again modeled in order to develop an algorithm to control the tail for climbing assistance. As discussed in Section 2.1.2, high centering is the common failure for similar surf-zone designs. In order to analyze this high centering condition MONTe was modeled climbing a six inch step, Figure 4.6. This screen shot from WM2D includes control parameters and output data. Figure 4.7 highlights the pitch data of the main body during the stalled climb. The slope of the pitch graph corresponds to the rate of change of that pitch, which nearly drops to zero when MONTe becomes high centered. This phenomenon is used to invoke the tail for assistance.

The control algorithm, Appendix A.1 and A.2, was then developed in MATLAB based on the adverse pitch rate during high centering. This control method actuates the tail in the event MONTe develops a high average pitch rate ($> 4°$/sec) followed by an low instantaneous pitch rate ($< 1°$/sec). MONTe was then modeled on the same six inch step, but now linked to the new control algorithm in MATLAB. Figure 4.8 shows MONTe successfully climbing the previous obstacle. This algorithm was then programmed onto the Monkey board for real obstacle testing.

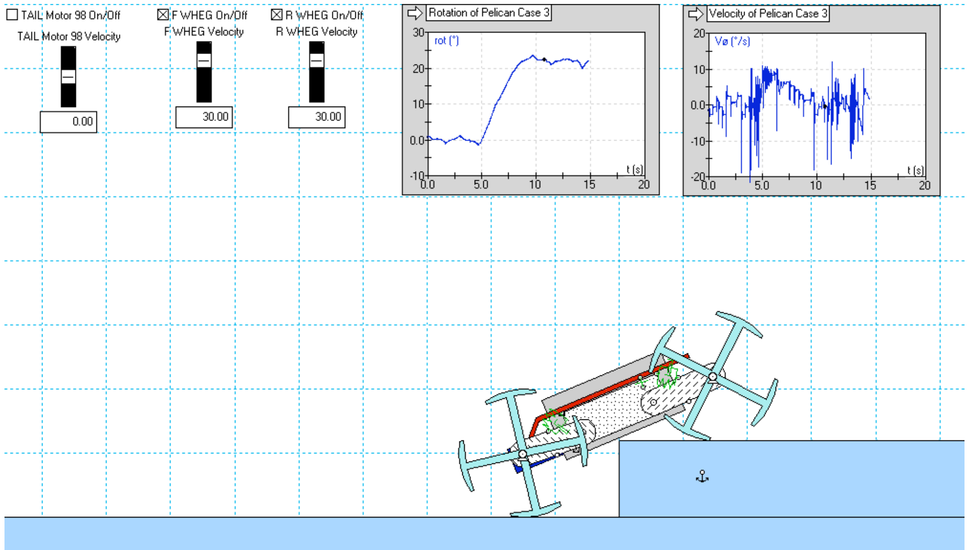

Figure 4.6: Model of MONTe encountering a high center condition

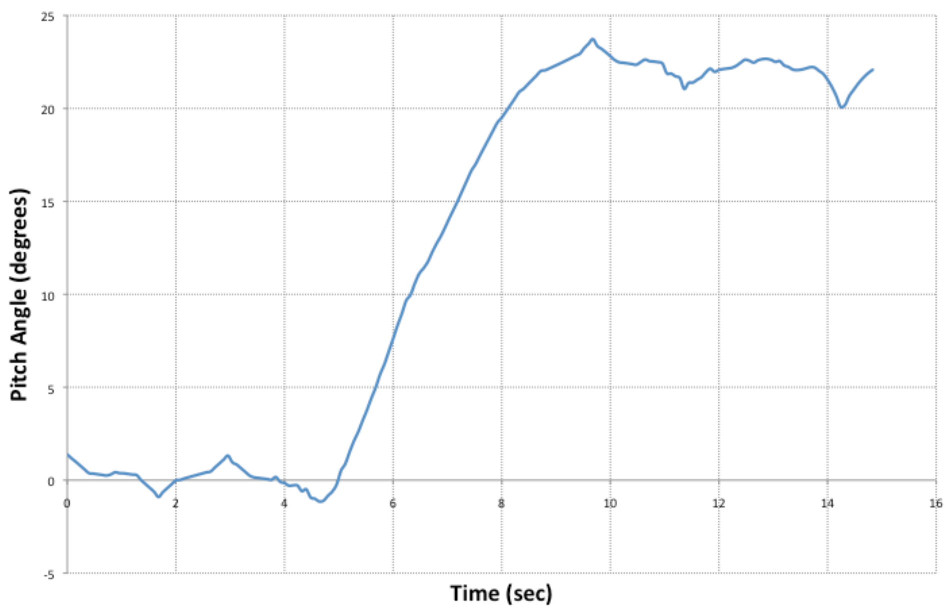

Figure 4.7: Pitch data from a high center scenario

Since the tail control algorithm was written in MATLAB based on WM2D, some modifications were necessary to host the algorithm on the Monkey board. Migrating the control program from MATLAB to the Monkey in C language required two major changes. First, the program would output desired tail angle vice tail speed. WM2D does not contain servo motors as devices and only allows control of rotational motors through torque, speed, or position values. When using the position method it would instantaneously move the tail to the new prescribed angle. In

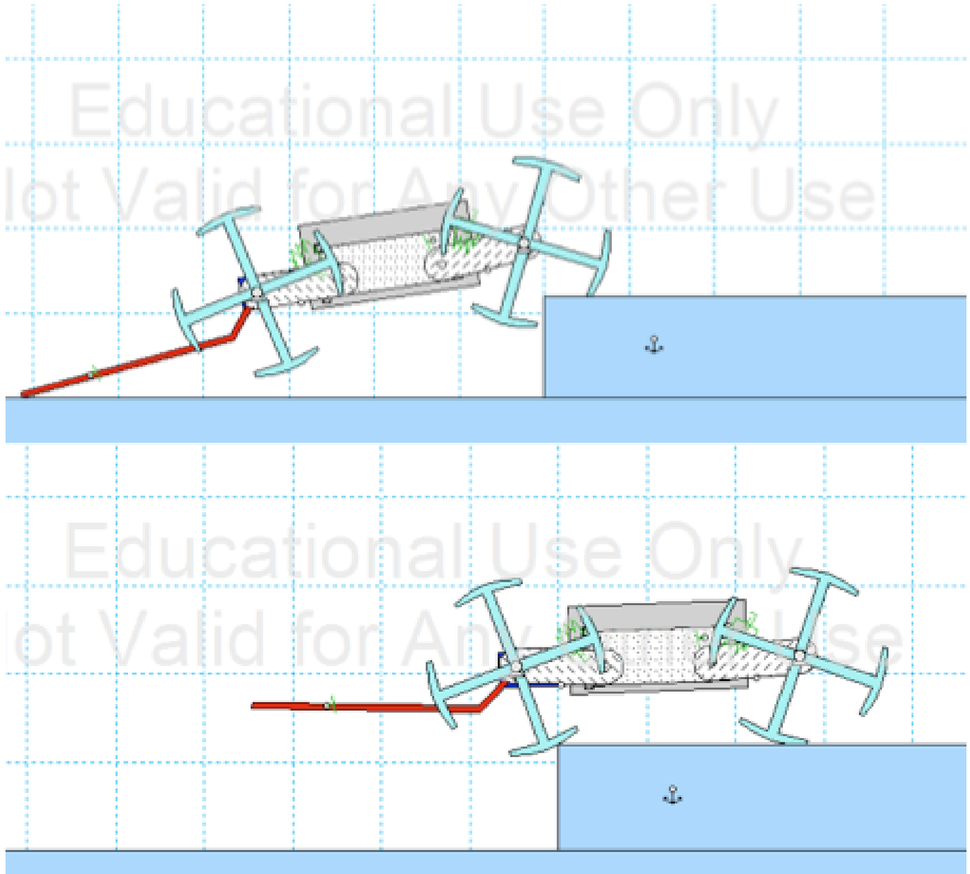

Figure 4.8: MONTe, in WM2D, interfaced with MATLAB tail control algorithm to overcome an obstacle

order to overcome this inaccuracy, tail speed was controlled vice angular position. For the real servo, the desired angle is prescribed through PWM and the inherent servo electronics control the actual speed to arrive at that angle. Therefore the output was revised to control tail position. Secondly, the real AHRS presents additional noise beyond that seen in the pitch rate of Figure 4.6. Since this noisy signal of raw pitch rate had the potential for creating erroneous tail control, the pitch rate was instead calculated from the pitch angle. The sampling rate for the pitch angle was set and used to calculate an average pitch rate. This time average was used in lieu of raw pitch rate and successfully suppressed the erratic signal. The Monkey control algorithm can be found in Appendix B.

Field Trials

Field trials were conducted with the modified control algorithm. MONTe was successful at actuating the tail when the front was lifted to simulate a stall condition. Additional testing was conducted while driving MONTe over obstacles manually. For these tests the tail was con-

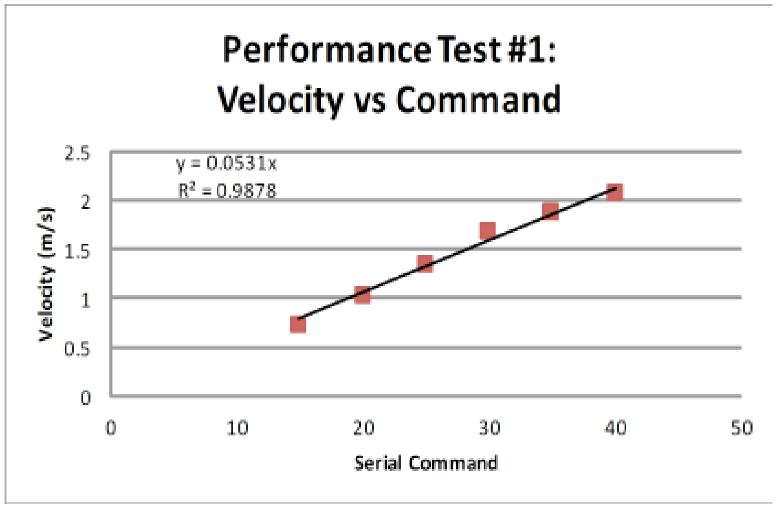

Figure 4.9: Performance curves over concrete

trolled autonomous by the Monkey board. MONTe repeatedly failed to climb step obstacles after multiple attempts. During this testing phase, several design issues were noted. The suspension system was not stiff enough and caused MONTe to travel low as it reached an obstacle. The Whegs™ were also not equipped with any form of traction and prevented MONTe from advancing onto the obstacle once the tail was lowered. Additional testing will be required to prove MONTe's overall design and in order to tune the tail control algorithm, see Section 5.1.

4.3.3 Speed of Advance

MONTe was tested over concrete to develop the motor control algorithms. Data was taken over a full range of control signals for forward velocity. Results are detailed in Figure 4.9.

Maximum velocity was determined to be 3.4 m/s for simplified serial command of 64. A nominal traveling velocity of 1.6 m/s was selected (serial command: 30). The maximum turning differential (without slipping) was 10.

4.4 ROS Testing

ROS testing explored the capabilities of the operating system in manual control and autonomous navigation. MONTe's program was successfully able to operate in full capacity. Figure 4.10 shows the current architecture using the `rxgraph` utility provided by ROS. Each active node is displayed with the connecting topics between nodes illustrated as well.

Each connection represents a subscribe, or publish path through the labeled topic. This pro-

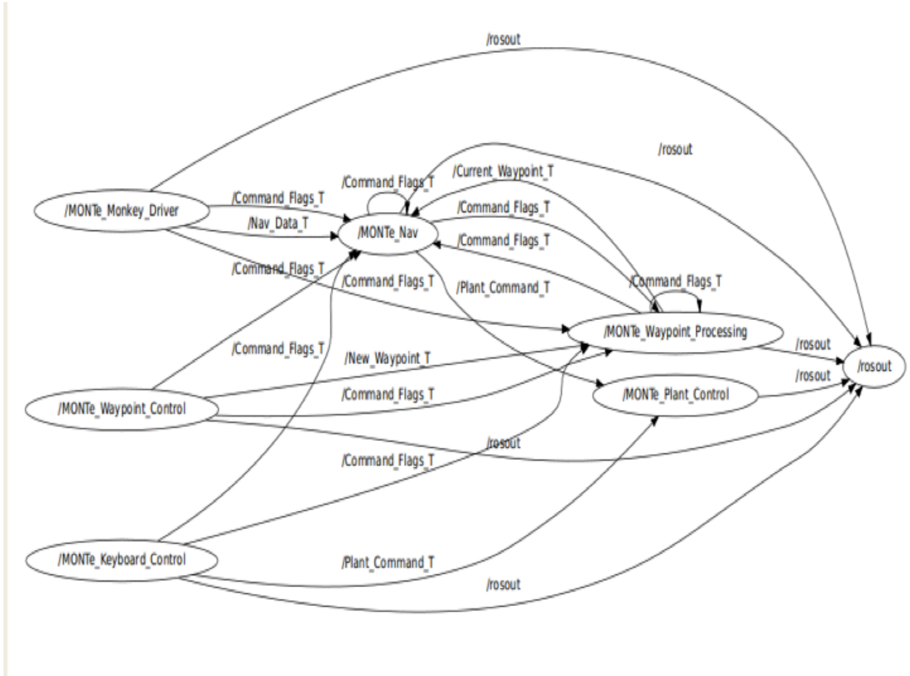

Figure 4.10: Output of ROS `rxgraph` function showing MONTe's node architecture

vides an easy display of the dependencies of each node. The Waypoint_Control, and Keyboard_Control were both successful in transmitting user commands to the rest of the system.

4.4.1 Remote Control

Remote operation of MONTe used a VNCserver for remote access during operation. The tests were successful as different program nodes could be started or stopped over a wireless network. For example, MONTe could be run in manual control only by starting ROScore, Keyboard Control, and Plant Control nodes. This also allowed changing and recompiling ROS nodes during testing. This proved useful during calibration of plant control constants.

MONTe could be remotely piloted as well. By running the nodes Keyboard_Control and Plant_Control, MONTe could be given commands to go forward, reverse, turn left, and turn right. The forward and reverse velocities, and turning rates could be adjusted.

Unacceptable latency was encountered during remote operation. Lag times of upwards of 15–30 seconds were observed in processing commands and receiving program responses while on VNC. MONTe would be unresponsive to commands during these periods of lag. This was not an issue during autonomous navigation, but did pose a significant issue when in manual control. Latency was not unexpected, though, and will drive future efforts to perform most control via Java-based GUI. Relegating the VNC to providing means of deep modifications of the operational program will relieve the processor and optimize performance.

A secure shell protocol (SSH) is another option for remote operation. This only brings up a unix shell for interacting with the system, so has less overhead associated with it. Multiple SSH connections could be made to bridge the gap until a more sophisticated interface can be implemented.

CHAPTER 5:
Future Work

5.1 Improvements to an Autonomous Tail

5.1.1 Tail Design

While modeling MONTe and conducting field tests, several design aspects were considered problematic. In one scenario the cross member for the tail protrudes too far below the chassis height. This limited ground clearance caused a unanticipated stall condition and required cycling the tail in order to free the tail. The cross member was originally included in the design in order to add rigidity to the tail and prevent it from twisting. Further testing should be conducted on the tail to either relocate or remove the cross member from the design. Additionally it may be possible to remove any support between the two sides of the tail and instead drive them as individual units. This would no longer require tuning the servo drive mechanisms to ensure that they are operated in tandem. This would also allow the tail to be controlled in a manner to induce a roll change as well as a pitch change.

Some testing scenarios also demonstrated that the friction at the end of the tail sometimes limited the forward motion of the robot. If the tail was invoked to assist with climbing on soft terrain, there was a chance that the tail would become almost embedded in the ground as it acted to lift the rear of the robot. The limited surface area of the tail tip and the high friction could cause a stall. It may be possible to reduce the susceptibility to this condition by adding a component to the tail that would instead roll along the ground. This concept would reduce the friction at the tail end and allow the traction from the Wheg™ to pull the robot in the desired direction of travel. Future testing should investigate the multitude of these design improvements.

5.1.2 Tail Control Algorithm

Currently the tail control program only uses the pitch data from the Monkey's CHIMU module to detect a high center condition and to actuate the tail. Additional sensors can be used to more accurately anticipate and detect a high centering condition. One would be to measure the force exerted on a Wheg™. When that force goes to zero, the Wheg™ can be considered unloaded. This can be easily sensed by limit switches attached to the suspension inside the drive assemblies. Also, the CHIMU's accelerometer data can be used to determine if forward motion has ceased, indicating a stall condition. This would require interfacing with plant control to ensure that the stall is not intended.

5.2 Improvements to Program Architecture

The current architecture will serve as the framework for future development of MONTe. Many ROS concepts have been proven and implemented such as the publisher/subscriber method of intermodal communications. The successful integration of sensors and hardware such as the Monkey, and the Sabertooth2x12 shows the usefulness of ROS for developing robotic projects.

Numerous features can be improved or added to in the existing architecture. The current code was developed using concepts from both C and C++. The overall architecture is object-oriented. Nodes are essentially "objects" that perform task, and communicate with others. Some of the legacy code utilized in development from previous projects was written in C, and is therefore not object-oriented. A traditional, linear based programming can easily be at the mercy of bugs that appear in the system, and its nature decreases its "portability". Portability allows code to be used in a variety of uses, and is one of the main goals of ROS. Therefore, the code needs to be modified in accordance with the principles of object oriented programming.

The goal would be then to modify, and conduct partial rewrites of the individual nodes themselves. This would bring the code in line with the principle of object oriented programming.

5.3 Navigation, Mapping, and Object Avoidance

Future work in navigation, mapping and object avoidance will include:

1. Implement additional sensors to improve navigation and autonomy.
2. Test navigational algorithms to optimize performance.
3. Utilize stereovision for object localization and avoidance.

Proposed sensors would include laser range-finders, acoustic sensors, and stereovision.

Previous work by Baravik [24] developed algorithms to conduct edge-detection and subsequent ranging. His process started by identifying the contours in the scene. These contours are then correlated to pick out the objects. The range is then determined by getting an angle to pixel of highest correlation by finding the pixel separation. The code was not implemented for real-time detection due to limitation of the camera and microprocessor. MONTe, however, has greater computational capacity in the LPC-100. Research could be done to develop a ROS node that could image the environment and process the results for object detection[24].

Path planning goes hand in hand with object detection. A simple algorithm is the “bug” algorithm. MONTe determines the heading to the current destination and drives the path until discovery of an obstruction. MONTe could then follow the perimeter of the obstacle until clear. The robot would then resume its heading to the destination [14].

A more sophisticated option would be to actively map the operating environment. MONTe would input navigational and sensor data to create a potential map that assigns strengths to the terrain it encounters. An algorithm could then be developed to determine the optimal path using Lagrangian or energy conservation techniques. This method could utilize MONTe’s terrain capabilities to drive over rough terrain if necessary as opposed to avoiding all terrain [14].

5.4 Remote Operation

One of the primary issues currently confronting the design is the rudimentary user interface. While effective for initial testing of the operational program, more sophisticated testing will require a more sophisticated interface. Such a program would need to be able to load waypoint routes, read sensor data, allow for remote control of MONTe, and a host of other useful functions.

The legacy NPS GUI displays: 1) positional data, 2) a location chart, 3) manual control panel, 4) waypoint routes. Additional features should include system status metrics for the power bus, environment, and other data of interest. Graphical displays could be expanded to include using of Google Earth™ to make the program suitable for general use. A display function for the potential map (filtered and unfiltered) would be useful to monitor the effectiveness of MONTe’s mapping process.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] T. Dunbar, "Demonstration of Waypoint Navigation for a Semi-Autonomous Prototype Surf-Zone Robot," Master's thesis, Naval Postgraduate, 2006.

[2] A. S. Boxerbaum, M. A. Klein, R. Bachmann, R. D. Quinn, R. Harkins, and R. Vaidyanathan, "Design of a semi-autonomous hybrid mobility surf-zone robot," *2009 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 974–979, Jul. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5229713

[3] C. L. Holland, "Characterization of Robotic Tail Orientation as a Function of Platform Position for Surf-Zone Robots," Master's thesis, Naval Postgraduate School, 2009.

[4] G. Dudek, P. Giguere, C. Prahacs, S. Saunderson, J. Sattar, L.-a. Torres-Mendez, M. Jenkin, A. German, A. Hogue, A. Ripsman, J. Zacher, E. Milios, H. Liu, P. Zhang, M. Buehler, and C. Georgiades, "AQUA: An Amphibious Autonomous Robot," *Computer*, vol. 40, no. 1, pp. 46–53, Jan. 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4069194

[5] R. Ding, J. Yu, Q. Yang, X. Hu, and M. Tan, "Platform-level design for a biomimetic amphibious robot," *2008 IEEE International Conference on Robotics and Biomimetics*, pp. 977–982, Feb. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4913132

[6] R. Quinn, J. Offi, D. Kingsley, and R. Ritzmann, "Improved mobility through abstracted biological principles," *IEEE/RSJ International Conference on Intelligent Robots and System*, vol. October, pp. 2652–2657, 2002. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1041670

[7] T. A. Williamson, "Modeling and Implementation of PID Control for Autonomous Robots," Master's thesis, Naval Postgraduate School, 2007.

[8] C. Bernstein, M. Connolly, M. Gavrilash, D. Kucik, and S. Threatt, "Demonstration of Surf Zone Crawlers : Results from AUV Fest 01 Report," Naval Surface Warfare Center, Tech. Rep., 2002.

[9] H. Zhang, T. P. Sattar, H. E. Leon-rodriguez, and J. Shang, *Climbing and Walking Robots: Towards New Applications*. InTech, 2007.

[10] J. R. Wilson, “Driving Force: DARPA’s Research Efforts Lead to Advancements in Robotics and Autonomous Navigation,” pp. 45–57, Dec. 2008. [Online]. Available: www.darpa.mil/WorkArea/DownloadAsset.aspx?id=2564

[11] K. Conley, “ROS/Introduction - ROS Wiki,” 2011. [Online]. Available: http://www.ros.org/wiki/ROS/Introduction

[12] M. Guenther. (2011, May) Robots using ros. [Online]. Available: http://www.ros.org/wiki/Robots

[13] J. Rice, B. Creber, C. Fletcher, P. Baxley, K. Rogers, K. McDonald, D. Rees, M. Wolf, S. Merriam, R. Mehio, J. Proakis, K. Scussel, D. Porta, J. Baker, J. Hardiman, and D. Green, “Evolution of seaweb underwater acoustic networking,” in *OCEANS 2000 MTS/IEEE Conference and Exhibition*, vol. 3, 2000, pp. 2007 –2017 vol.3.

[14] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005. [Online]. Available: http://books.google.com/books?id=S3biKR21i-QC\&pgis=1

[15] M. J. Willis, “Proportional-Integral-Derivative PID Controls,” *Chemical and Process Engineering*, pp. 1–13, 1999.

[16] K. Johan and R. M. Murray, *Feedback Systems An Introduction for Scientists and Engineers University of California Santa Barbara*. Princeton University Press, 2007.

[17] M. Slatt, “Mechanical Design of MONTe: A Semi-Autonomous Surf-Zone Prototype Robot,” In Preparation, Naval Postgraduate School, 2011.

[18] Servo city. [Online]. Available: http://www.servocity.com/html/spg400a_top_mount.html

[19] W. E. Snyder, *Industrial Robots: Computer Interfacing and Control*. Englewood Cliffs, New Jersey: Prentice-Hall, 1985, vol. Chapter 4.

[20] K. Conley. (2011, 5) Ros concepts. [Online]. Available: http://www.ros.org/wiki/ROS/Concepts

[21] *Sabertooth2x12 Users Guide*, Dimension Engineering, 2010. [Online]. Available: http://www.dimensionengineering.com/datasheets/Sabertooth2x12.doc

[22] J. Homppi, "Programming RS232 devices using LINUX." [Online]. Available: http://www.ontrak.net/linux.htm

[23] *Working Model 2D Users Guide*, Design Simulation Technologies. [Online]. Available: http://www.workingmodel.com

[24] K. A. Baravik, "Object Localization and Ranging Using Stereo Vision for Use on Autonomous Ground Vehicles," Master's Thesis, Naval Postgraduate School, 2009.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Simulation Tail Control Code

A.1 MATLAB Control Algorithm

This controls MONTe's tail and is used in conjunction with Working Model 2D and MATLAB.

```
%-----------------------------------------------------------------------
% MONTe TAIL CONTROL
% Version 2.0
% Written by Steven Halle
% Thesis:  Design and Implementation of a Semi-Autonomous Surf-Zone Robot
% using Advanced Sensors and a Common Robot Operating System
% 30MAY2011
%-----------------------------------------------------------------------
% Details for interfacing WM2D with MATLAB are found in WM2D user guide

function tail_spd = TAIL2(body_rate,body_angle,tail_angle)
%  values inputted from WM2D
% (pitch rate, pitch angle, tail angle)
%  outputs the tail motor speed

% declares variables as global to hold values after the function is called
global ab               % shorthand for angle of body rate (pitch rate)
global climb            % 1 or 0 to indicate tail assitance needed
global avg_rate         % holds the value for the averge pitch rate

k=size(ab,2);               % loop to keep most recent pitch rate in ab
for n=1:k-1
   ab(1,k-n+1)=ab(1,k-n);
end
ab(1,1)= body_rate;         % logs current body rate as first in history
avg_rate=mean(ab,2);        % takes average of entire ab array

ab_new=ab(1,1:6);
new_rate=sum(ab_new)./6;    % local variable, average newest 6 pitch rates

% CONTROL LOGIC
if(avg_rate > 4) && (new_rate < 1)      % presents a stall climb situation
    climb=1;                            % sets climb mode
    tail_spd=60;                        % actuautes the tail
elseif (body_angle > 5) && (climb==1)   % continues tail until robot levels
    tail_spd=60;
elseif(body_angle < 5.0) && (tail_angle > 145.0)  % stows the tail
    tail_spd=-60;
    climb=0;
else
    tail_spd=0;
```

```
end
```

A.2 MATLAB Initialization

This initializes MATLAB in order execute Appendix A.1.

```
%-----------------------------------------------------------------------------------------
% MONTe TAIL CONTROL INITIALIZATION
% Version 2.0
% Written by Steven Halle
% Thesis:  Design and Implementation of a Semi-Autonomous Surf-Zone Robot
% using Advanced Sensors and a Common Robot Operating System
% 30MAY2011
%-----------------------------------------------------------------------------------------

% This code is to be used in conjunction with Working Model 2D and MATLAB
% Details for interfacing WM2D with MATLAB are found in WM2D user guide

% Enables MATLAB proxy to interface with external application
enableservice('AutomationServer',true);

% Initializes global vairables used in control program and sets value to 0
global climb avg_rate ab
ab=zeros(1,100);
avg_rate=0;
climb=0;
```

APPENDIX B:
MONTe Tail Control Code for Monkey Board

This code is an edited portion of a Ryanmechatronics suite that the Monkey Board hosts for autonomous tail control of MONTe.

```
/*******************************************************************
MODULE:     UserMode.c
VERSION:    1.00
CONTAINS:   Specific commands and modes for user actions
COPYRIGHT:  Ryan Mechatronics
Date:        Dec. 2009
*******************************************************************/

/*******************************************************************
INSTALLED CODE FOR AUTONOMOUS TAIL CONTROL
Edited by    Steven Halle
Thesis:      Design and Implementation of a Semi-Autonomous Surf-Zone Robot
             using Advanced Sensors and a Common Robot Operating System
Date:        30MAY2011

NOTES:     This code is provided as part of suite from Ryan Mechantronics.
Portions of the supplied code were edited to create a tail control algorithm.
The majority of code pertaining to MONTe resides in this file.  Additional
changes can be found in the errata section below
*******************************************************************/
/*******************************************************************
Additional Change Section

pwm.c   C:\...\Common\Public\PWM    3 occurrences
  Line 38    //Edit by Steve Halle -
         only servos 5/6 will be used for MONTe
  Line 61    //Edit by Steve Halle -
         Sets configuration for Servos 5/6 according to MONTe design
  Line 118 //Edit by Steve Halle -
         ensures that servo does not get passed value outside the range of servo

pwm_uplink.c   C:\...\Common\Public\PWM    5 occurrences
  Line 15    static int bRC_Uplink_Active = FALSE; //Edit by Steve Halle -
         unsure if needed to invoke uplink
  Line 49    //Edit by Steve Halle -
         changed the function call in order to include input channel parameter
  Line 52    //Edit by Steve Halle -
         removed previous argument type, must now pass SERVO_IN_1, etc
  Line 100   //Edit by Steve Halle -
         removed previous argument type, must now pass SERVO_IN_1, etc
  Line 107   //Edit by Steve Halle -
         removed previous argument type, must now pass SERVO_IN_1, etc
```

```
control.c  C:\...\Platforms\Public\Generic   7 occurrences
  Entire file Removed from Project  //Edit by Steve Halle -
        removed since PID not necessary for servo control

projectconfig.h  Monkey_User_Sandbox_SWD   1 occurrence
  Line 137  #define CFG_CHIMU_ORIG  //Edit by Steve Halle -
        use autoselect is unknown, else orig as of 4/2011

*********************************************************************/

#include "globals.h"
#include "uart.h"
#include "lpcUART.h"
#include "math.h"
#include "iap.h"
#include "string.h"
#include "util.h"
#include "system.h"
#include "main.h"
#include "CommOutput.h"
#include "UserMode.h"
#include "events.h"
#include "adc.h"
#include "spi.h"
#include "gps.h"
#include "stdio.h"
#include "navigation.h"
#include "guidance.h"
#include "control.h"
#include "sd_logger.h"
#include "pwm.h"          //Edit by Steve Halle -
                                            //      need add pwm.h since control() removed
#include "pwm_uplink.h"   //Edit by Steve Halle -
                                            //      allows pass thru PWM signals

#include "..\..\Private\User_Library_Functions\UserFunctions.h"

//#include "..\..\..\..\Common\Public\FFT\dsplib_testbench_fft_main.h"

#ifdef CFG_INSFILTER
  #include "ins_filter.h"
#endif
#ifdef CFG_BMP085
  #include "bmp085.h"
#endif


unsigned char user_mode = 0;   //State machine status
// Nav and guidance variables
NAVIGATE nav_sol;
GUIDANCE guide_sol;
```

```
//Edit by Steve Halle
//Global Variables declared outside user function loops

float body_angle;        //This hold value for the angle of the pelican case of the robot
float b_a_prev;          //Body Angle previous
float body_rate;         //This holds value for the rate at which the body_angle is changing
float avg_body_rate;     //This holds value for the average rate of body change
float b_r_history[40];   //This holds a history of the last 40 body_rate
                                                 //(2 seconds worth if executed every 50ms)
float tail_output;       //This holds value for the desired tail position
int   climb;             //This holds status for the need for tail to take climbing action

void User_Init( void)
{
  Led_Off(LED_RED);
  //Edit by Steve Halle - below code is copied from control.c to eliminate autopilot
  //-----------------------------------------------------------------------------------
     int i;
    //INIT PWM base
    //Change this to PWM_10MSEC_BASE for 100 Hz updates to servos
    PWM_Init(PWM_20MSEC_BASE);
    //Initialize outputs
    Servo_Config();

    //The below may be moved.
        //Servo values should be updated to real initial values before PWM is started
        //to make sure no startup glitch occurs

    //Start servos.  Future Servo_Set calls will just update value for a particular channel
        //or a general update on all of them

        PWM_Start();

  //------------------------------end EDIT by Steve Halle ------------------------------
}

void User_Main(void)
{
  //Main loop
  // Note that this is the time / tasking loop.
  //There are protected areas in this section you should leave alone.

  // State machine for task processing by user is in user_process()
  //

  //INIT Guidance, Nav and Control Modules (including PWM's)
  //NOTE:  Waypoint has been commented out until we get IAP going on Cortex
  Waypoint_Init();
  Navigate_Init();

  //Edit by Steve Halle - removed control_init() and Guidance_Init() since autopilot not used
```

```
//------------------------------------------------------------------------------------------
// Guidance_Init(&guide_sol);
// Control_Init();
//----------------------------------------end EDIT--------------------------------------------

#if defined CFG_SDCARD
  bLoggerOK = sd_init(); //Init SD card logger if enabled
#endif

gThruputCnts = 0;

User_Init();


//Start processing loop
while (1)
{
  gThruputCnts++;

   //Pseudo-tasking occurs here
  if (time_1ms_flag) // 1ms (1000 Hz) tasks
  {
      /*--------------------------------------------------------------------------------*/
      /* BEGIN PROTECTED --- DO NOT REMOVE --- UNIT WILL NOT OPERATE CORRECTLY */
          //Handle serial port receipt
          Main_SerialPort_Process();
                      //Handles double buffering of input from serial coms

                      //Parse now that the interrupts seem over (RX FIFO has been emptied)
          SSP1_Parse_CHIMU();
          //Parse GPS
          GPS_Parse();
          //Process uplinked waypoints - Event handler will indicate if there
                      //is a complete set waiting for you
          Waypoint_Process();
          //INS filter processing
          #ifdef CFG_INSFILTER
          INS_Filter_Process();
          #endif
          #ifdef CFG_BMP085
          Baro_Process(1000); //Updates pressure sensor at a 1 second rate.
                      //Maximum update rate is about 40 msec
          #endif
      /* END PROTECTED ------------------------------------------------------------------*/
      /*--------------------------------------------------------------------------------*/

      User_Process(); //Handles user processing (like mode switches, etc)

      time_1ms_flag = 0;//Clear time tick
  }

  if (time_5ms_flag) //5ms (200 Hz) tasks
```

```
{
    //No Tasks
    time_5ms_flag = 0;//Clear time tick
}


if (time_10ms_flag) //10ms (100 Hz) tasks
{
    //No Tasks
    time_10ms_flag = 0;//Clear time tick
}


if (time_20ms_flag) //20 ms (50Hz) tasks
{
      /*-----------------------------------------------------------------------------*/
      /* BEGIN PROTECTED --- DO NOT REMOVE --- UNIT WILL NOT OPERATE CORRECTLY */

          //Control_Process(&guide_sol);
                          //Handle control functions (servos) at 50Hz
          //Edit by Steve Halle
                      //removed as a test to eliminate
                          //control functions for built in autopilot

          #ifdef CFG_USE_MONKEY_TELEMETRY
          TX_Com_Process(); //Handles output messages
          #endif
      /* END PROTECTED ---------------------------------------------------------------*/
      /*-----------------------------------------------------------------------------*/
    time_20ms_flag = 0;//Clear time tick
}

if (time_100ms_flag) //100ms (10 Hz) tasks
{
      /*-----------------------------------------------------------------------------*/
      /* BEGIN PROTECTED --- DO NOT REMOVE --- UNIT WILL NOT OPERATE CORRECTLY */
          ADC_Process(); // 10 Hz, start a burst ADC read.  Global gADC holds result.
          gOK_TO_SEND = TRUE;  //DEBUG - Spits CHIMU returning messages back
      /* END PROTECTED ---------------------------------------------------------------*/
      /*-----------------------------------------------------------------------------*/

    //Below is for special output on UART 1
    //User_NMEA_Output();

    //Below is for SD card logging of standard data set if card is present
    //Called at 10 Hz, but only writes to disk after 10 entries
    #if defined CFG_SDCARD
    SD_StandardLogging(TRUE);
    #endif


    time_100ms_flag = 0; //Clear time tick
```

```
    }

    if (time_1000ms_flag) //1000ms (1 Hz) tasks
    {
        System_Check_CPU();  //Checks CPU load and puts it into Monkey output message as needed

        //Waypoint_demo();

        //Edit by Steve Halle - passes GPS posit from spare port in NMEA output every 1 sec
        User_NMEA_Output();
        time_1000ms_flag = 0; //Clear time tick

    }
  }//Loop back around

}






void User_Process(void)
{
  //This is where mode specific actions should happen.
  // It is where most of your decision making occurs

  static unsigned long lasttime = 0;
  static unsigned long elapsed_time = 0;
  unsigned long dt_msec = 0;

  dt_msec = getTimeCounts() - lasttime;
  lasttime = getTimeCounts();
  elapsed_time += dt_msec;


  User_Event_Process(); //Go check for events that may have occurred (user event messages)

  switch (user_mode)
  {
    case(0):
                //First two cases are for testing and debugging purposes

                //Example mode 0
                if(elapsed_time > 1000)
                {
                  uart0Puts("Here I am\r");
                        Led_Flash(LED_RED,1);
                        user_mode++;
                        elapsed_time = 0;
                }
      break;
```

```
case(1):
          //Example mode 1
          if(elapsed_time > 1000)
          {
                  Led_Flash(LED_RED,2);
                  user_mode++;
                  elapsed_time = 0;
                  //servo_out[SERVO_RIGHT_6].value = 110;   //for debugging servo output
          }

  break;
 case(2):
          //Example mode 2

          if(elapsed_time > 200)  //loop executed every 200 msec
                                  //IMPORTANT: calculations below use this time step
          {

                  int i;
                  float tot_b_r=0;
                  for (i = 40; i>0; i--)
                  {
                    b_r_history[i-1]=b_r_history[i-2];
                                                  //performs a shift in the history array
                                                  //- b_r_history[0] remains value zero
                  }
                  for (i = 1; i<40; i++)
                  {
                    tot_b_r=tot_b_r + b_r_history[i];
                                          //calculates the sum of all the
                                          //b_r_history elements
                  }
                  avg_body_rate=tot_b_r/39;
                                          //calculates the average body rate based on the
                                          //last 39 array elements (b_r_history[0]=0)
                  b_a_prev=body_angle;
                                          //assigns the previous body_angle
                  body_angle = gAttitude.euler.theta;
                                          //sets body angle to current pitch angle of
                                          //Monkey (in units radians)
                  b_r_history[0] = (body_angle - b_a_prev)/0.20;
                                          //calculates rate based on 50ms calc steps


                  if(servo_in[0].ro_msec > 1.5)
                  //servo_in[1] matches SERVO_INPUT_2
                    {
                    Led_On(LED_RED);
                  //RED LED on Monkey means board is in manual control
                    PWM_Uplink_Passthru( SERVO_INPUT_2 , SERVO_RIGHT_6 , FALSE );
                  //input channel, output, failsafe -- see prototype
                    PWM_Uplink_Passthru( SERVO_INPUT_3 , SERVO_LEFT_5 , FALSE );
```

```
                    //input channel, output, failsafe -- see prototype
                    }
                    else
                    {
                        //Autonomous Mode
                        // TAIL LOGIC
                        Led_Off(LED_RED);
                        if((avg_body_rate > 0.045) && (b_r_history[0] < 0.035))
                        {
                        //sense high center condition
                        climb = 1;
                        servo_out[SERVO_RIGHT_6].value = 225;
                        servo_out[SERVO_LEFT_5].value = 225;
                        Servos_Update_All();
                        }
                        else if((body_angle > 0.09) && (climb==1))
                        {
                        //continue to hold tail until climb complete
                        servo_out[SERVO_RIGHT_6].value = 225;
                        servo_out[SERVO_LEFT_5].value = 225;
                        Servos_Update_All();
                        }
                        else    //(body_angle < 0.09)
                        {
                        //return to neutral position
                        climb=0;
                        servo_out[SERVO_RIGHT_6].value = 115;
                        servo_out[SERVO_LEFT_5].value = 115;
                        Servos_Update_All();
                        }

                    }


                    elapsed_time = 0;
                }
        break;
// Edit by Steve Halle

    }
}



int User_Event_Process( void )
{
  User_Uplink_Msg cmdmsg;

  switch(Event_Retrieve( &gEvent))
  {
    case -1:
      return (-1);
```

```
      break;
    case 1:
      //Event has been loaded into global, now figure out what to do with it
      if (gEvent.id != EVT_UPLINK_MSG) return (-1);
                  //This ignores other uplink messages,
                  //like waypoints, etc... that are handled by the main, common processing
                memmove( &cmdmsg, &gEvent.payload, gEvent.length);
      // If a mode switch, take it and leave
      if (user_mode != cmdmsg.mode)
      {
        user_mode = cmdmsg.mode;
        return (1);
        break;
      }
      //
      //For example message, we have a command word that
                  //indicates what command has been sent:
      //

      switch(cmdmsg.action)
      {
        case USER_ACTION_REQUEST_STATUS:
          User_Status_Output();
          break;
        case USER_ACTION_ABORT:

          break;
        case USER_ACTION_CMD_ANGLES:
          guide_sol.att_des.euler.phi =         cmdmsg.phi_desired;
          guide_sol.att_des.euler.theta =       cmdmsg.theta_desired;
          guide_sol.att_des.euler.psi =         cmdmsg.psi_desired;
          break;

        return(1);
        break;
      default:
        return(-1);
        break;
    }
  }
}


unsigned char User_Status_Output( void )
{
        // This is a message wrapper for special user messages.
        //Normal telemetry contains most items of interest
        //  This message wraps output in the user message format 0xAA
        //
  int index = 0;
  short int sint_tmp = 0;
  float ftmp = 0.0;
```

```
  int i = 0;
  if(gUserBytesOut.isbusy == TRUE) return (FALSE);
        //Buffer hasn't been transmitted yet, don't respond yet
  // Example output message is:
  // Modes / status
  // Timer / counters
  // Example float data

   //
  //Modes
  gUserBytesOut.payload[index]= 0x01; index++;
//User message ID - chose 0x01 for no good reason
  gUserBytesOut.payload[index]= 0x00; index++;
//User message length (overwrite at bottom once index is summed up)
  gUserBytesOut.payload[index]= user_mode; index++;
//Local state machine
// Timers / counters
  memmove(&gUserBytesOut.payload[index],
                 &gTime_Msec, sizeof(unsigned long));
                        index += sizeof(unsigned long);
        //System running time in milliseconds
  memmove(&gUserBytesOut.payload[index],
                 &gThruputHz, sizeof(unsigned long));
                        index += sizeof(unsigned long);
        //Thruput
  // Example float data
  ftmp = (99.0);
  memmove(&gUserBytesOut.payload[index], &ftmp,
                 sizeof(float));
                        index += sizeof(float);
  // Done now with populating

  // Now, replace the length byte with our total index to help decoding by a ground station
  gUserBytesOut.payload[1]=index; //No bumping index here, we are just replacing a value
  gUserBytesOut.length = index;
  // Now, the global user bytes are all setup and ready to go
  // Flag it as full, then send it all out using the message 0xAA user bytes wrapper
  // When it is gone, the flag will be cleared
  gUserBytesOut.isbusy = TRUE;
  Tx_Com_Add_Special_Message(MSGOut_0xB0_User_1);
  return (TRUE);
}



unsigned char User_NMEA_Output( void )
{
  //Example output of various data in NMEA format on the spare UART
  //For details of GPS structure see globals.h
  int index = 0;
  int i = 0;
  char stemp[128];
```

```
  char pOutString[196];
  unsigned char  pFieldXSUM = 0;

  // Send Header first
  sprintf(pOutString,"$RM,");
  // Add other data
  //GPS first
  sprintf(stemp,"%01i,%081d,%081d,%031d,%041d,%041d,%041d,%lX,", (int)gGPS.satstracked,
                                   (long)(gGPS.latitude*10000),(long)(gGPS.longitude*10000),
                (long)(gGPS.altitude),(long)(gGPS.velN*10),
                                   (long)(gGPS.velE*10), (long)(gGPS.velD*10), (long)(gGPS.TOW));
  strcat(pOutString,stemp);
  sprintf(stemp,"%041d,%041d,%041d,%041d,%041d,%041d,%041d,%041d,%041d,",
                  (long)(gAttitude.euler.phi*100),
                  (long)(gAttitude.euler.theta*100),
                  (long)(gAttitude.euler.psi*100),
                  (long)(gSensor.rate[0]*100),
                  (long)(gSensor.rate[1]*100),
                  (long)(gSensor.rate[2]*100),
                  (long)(gSensor.acc[0]*100),
                  (long)(gSensor.acc[1]*100),
                  (long)(gSensor.acc[2]*100));
  strcat(pOutString,stemp);
#ifdef CFG_INSFILTER
  sprintf(stemp,"%081d,%081d,%031d",
                  (long)(gINS.latitude*10000),
                  (long)(gINS.longitude*10000),
                  (long)(gINS.altitude));
  strcat(pOutString,stemp);
#endif

  //OK, now we have the whole string.  Time to find the checksum
  for (i=1; i<strlen(pOutString);i++)
          //NOTICE:  Starting at 1, because xsum doesn't use $ in front
  {
    pFieldXSUM ^= pOutString[i];
  }
  //Tack it on the end
  sprintf(stemp,"*%02x", pFieldXSUM);
  strcat(pOutString,stemp);

  //Output the string
  uart1Puts(pOutString);

  //Output the <CR><LF>
  uart1Putch(0x0D);
  uart1Putch(0x0A);

  return (TRUE);
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
ROS Code

All electronic versions of this code will be incorporated into the upcoming NPS ROS Stack.

C.1 Navigation Node

Following code performs navigation for MONTe.

```
/***************************************************************************
Title:      MONTe_Nav 0.0 (ROS Node)

Author:     Jason Hickle

Purpose:  Node that allows MONTe to navigate through the world.
        Handles the following functions:
                1) Monitors Command_Flags to determine behavior.
                2) Pulls current position data (GPS and heading)
                3) Gets current waypoint
                4) Computes desired heading and range to waypoint.
                5) Computes plant commands using current and desired hdg.
                6) Updates Command_Flags and Plant_Control

Use:        Communication protocol handled via ROS messaging.  Launch program as
        part of roslaunch.

          Runs at loop rate of 4 Hz.

ROS Notes:
        Name-               "MONTe_Navigation"

        Publications-       "Plant_Command_T"
                            "Command_Flags_T"

        Subscriptions-      "Command_Flags_T"
                            "Nav_Data_T"
                            "Current_Waypoint_T"

        Messages-           Command_Flags.msg
                            Plant_Command.msg
                            Waypoint.msg

        Services-           None

Version History:

        ---- Version 0.0 ----
        Mar 21st, 2011
```

```
        LT Jason Hickle

        Enable Manual Control mode in main control loop.
*************************************************************************/

/*      Libraries                   */
#include <ros/ros.h>
#include <math.h>
#include "std_msgs/String.h"
#include "MONTe/Plant_Command.h"         // Message format (.msg)
#include "MONTe/Waypoint.h"              // Message format (.msg)
#include "MONTe/Command_Flags.h"         // Message format (.msg)
#include "MONTe/Nav_Data.h"              // Message format (.msg)

#include <sstream>

/*      Defines                     */

const double PI = 3.14159265;

const unsigned char STOP_R = 190;        // Right motor stop
const unsigned char STOP_L = 64;         // Left motor stop

const unsigned char R_CORRECT = 1;       // Correction factor to calibrate forward speed
const unsigned char L_CORRECT = 3;

const double RANGE_THRESH = 2.5;
const float  HDG_ERROR_THRESH = 3.0;

const char   FWD_SPEED = 30;
const char   MAX_TURN_DIFF = 10;

/*      CF holds all command flags. Expand as necessary.  Ensure publishing    /
/       utilizes default values for any flag unchanged to ensure flags are     /
/       not being changed unecessarily.                                        */
typedef struct
        {
                bool autonav;
                char mode;
                bool route;
        }CF;                                 //Define flags structure

/*      WP_P stores all data necessary to navigate to, and determine           /
/       behavior mode for a route.                                             */
typedef struct
        {
                double lat;
                double lon;
                char   action;
        }WP;                                 //Define waypoint structure

/*      NAVDATA is for current positional data                                 */
```

```
typedef struct
        {
                double lat;
                double lon;
                double  heading;
        }NAVDATA;                       //Define Current Position structure



/*      Global Variables        */
WP Current_WP;
CF Cmd_Flags;
NAVDATA Current_Nav_Data;

double K_P_coefficient;                 // Proportional constant for PID control
float right_command, left_command;

/*      Functional Prototypes   */
void Command_FlagsCallback(const MONTe::Command_FlagsConstPtr& flags);

void WaypointCallback(const MONTe::WaypointConstPtr& way_pt);

void Nav_DataCallback(const MONTe::Nav_DataConstPtr& nav_dat);

int Navigate();

int Plant_Compensator(double range, float desired_hdg, float current_hdg);

int main(int argc, char **argv)
{
/*      ROS Initializations     */
        ros::init(argc, argv, "MONTe_Navigation");  // Set up ROS node

        ros::NodeHandle n;  // set up handle for this node

                // Set up all publishers for node
        ros::Publisher plt_cmd_pub = n.advertise<MONTe::Plant_Command>("Plant_Command_T", 1);
        ros::Publisher cmd_flg_pub = n.advertise<MONTe::Command_Flags>("Command_Flags_T", 1);

                // Set up all subscriptions for node
        ros::Subscriber sub_cf = n.subscribe("Command_Flags_T", 1, Command_FlagsCallback);
        ros::Subscriber sub_wp = n.subscribe("Current_Waypoint_T", 1, WaypointCallback);
        ros::Subscriber sub_nav = n.subscribe("Nav_Data_T", 1, Nav_DataCallback);

        ros::Rate loop_rate(4);         // Sets 4hz cycle for main loop

                // Set up message handles for communicating with topics.
           MONTe::Command_Flags flags, pub_flags;
           MONTe::Waypoint way_pt;
           MONTe::Plant_Command cmd;
           MONTe::Nav_Data nav_dat;
```

```
        int nav_val = 0;
        Cmd_Flags.autonav = 0;

/*      Calculate Kp coefficient for turning. Define MAX_TURN above           */
        K_P_coefficient = pow((((double) MAX_TURN_DIFF)/80.0), (1.0/3.0));

/*      End ROS Initializations */
        while(ros::ok())
        {
        ros::spinOnce();        // Callback to all active topics

/*      Begin Navigation        */
        if(Cmd_Flags.autonav == 1)
        {
                nav_val = Navigate();

                if(nav_val == 1)        // Waypoint reached, inform and stop MONTe
                {
                        pub_flags.auto_nav = Cmd_Flags.autonav;
                        pub_flags.nav_mode = 'N';
                        pub_flags.incoming_route = Cmd_Flags.route;
                                // N indicates waypoint reached, need next waypoint
                        cmd_flg_pub.publish(pub_flags); // Publish to cmd_flags topic

                        cmd.left = (unsigned char) STOP_L; //Send stop command to PlantControl
                        cmd.right = (unsigned char) STOP_R; //while waiting for next waypoint
                        plt_cmd_pub.publish(cmd);
                }
                        else    // Send command signal to plant to move towards waypoint
                        {
                                cmd.left = (unsigned char) left_command;
                                cmd.right = (unsigned char) right_command;
                                plt_cmd_pub.publish(cmd);
                        }
        } // End Navigation

        loop_rate.sleep();      // Sleeps to maintain loop_rate
        } // end while loop

} //end main

/***********************************************************************
Function:       Command_FlagsCallback

Description:    Pulls waypoint from the Command_Flags_T topic

Parameter:      1) Pointer to ROS message from topic Command_Flags_T

Return Value:   None
***********************************************************************/
void Command_FlagsCallback(const MONTe::Command_FlagsConstPtr& flags)
{
```

```
  Cmd_Flags.autonav = flags-> auto_nav;
  Cmd_Flags.mode = flags-> nav_mode;
  Cmd_Flags.route = flags-> incoming_route;
} // end callback

/***************************************************************************
Function:        WaypointCallback

Description:     Pulls waypoint from the Current_Waypoint_T topic

Parameter:       1) Pointer to ROS message from topic Current_Waypoint_T

Return Value:    None
***************************************************************************/
void WaypointCallback(const MONTe::WaypointConstPtr& way_pt)
{
  Current_WP.lat = way_pt->latitude;
  Current_WP.lon = way_pt->longitude;
} // end callback

/***************************************************************************
Function:        Navigation_DataCallback

Description:     Pulls waypoint from the Current_Waypoint_T topic

Parameter:       1) Pointer to ROS message from topic Current_Waypoint_T

Return Value:    None
***************************************************************************/
void Nav_DataCallback(const MONTe::Nav_DataConstPtr& nav_dat)
{
  Current_Nav_Data.lat = nav_dat->latitude;
  Current_Nav_Data.lon = nav_dat->longitude;
  Current_Nav_Data.heading = nav_dat->heading;
} // end callback

/***************************************************************************
Function:        Navigate

Description:     Navigate MONTe.  Pulls current posit and waypoint.  Determines
        range and heading to desitiantion.  Calls plant control function to
        produces plant command.

Parameter:       None

Return Value:    0 - Success
***************************************************************************/
int Navigate()
{
        static double lat, lon, *lat_ptr, *lon_ptr;      // Current position
        static double wlat, wlon, *wlat_ptr, *wlon_ptr; // Waypoint position
        static double lat_diff, lon_diff, *lat_diff_ptr, *lon_diff_ptr;
```

```
                    // variables to calculate differences in latitude & longitude
        static double rng, *rng_ptr;        //Range (in yards)
        static float cur_hdg, new_hdg, *cur_hdg_ptr, *new_hdg_ptr;

/*      Pointer initializations */
        lat_ptr = &lat;
        lon_ptr = &lon;
        wlat_ptr = &wlat;
        wlon_ptr = &wlon;
        rng_ptr = &rng;
        lat_diff_ptr = &lat_diff;
        lon_diff_ptr = &lon_diff;
        cur_hdg_ptr = &cur_hdg;
        new_hdg_ptr = &new_hdg;

/*      Update variables        */
        *lat_ptr = Current_Nav_Data.lat;
        *lon_ptr = Current_Nav_Data.lon;
        *wlat_ptr = Current_WP.lat;
        *wlon_ptr = Current_WP.lon;
        *cur_hdg_ptr = Current_Nav_Data.heading;

/*      Compute range to current waypoint.      */
        *rng_ptr = sqrt((((2000 * (*wlat_ptr))-(2000 *
                        (*lat_ptr)))*((2000 * (*wlat_ptr))-(2000 * (*lat_ptr))))+
                                (((1600 * (*wlon_ptr))-(1600 * (*lon_ptr)))*
                        ((1600 * (*wlon_ptr))- (1600 * (*lon_ptr)))));

                if (*rng_ptr <= RANGE_THRESH) //When close enough to waypoint, action
                {                       //code takes effect and next waypoint is loaded
                        return 1;
                }

        // 3600 converts lat_diff and lon_diff to decimal seconds for accuracy
        *lat_diff_ptr = 3600 * ((*wlat_ptr)-(*lat_ptr));
        *lon_diff_ptr = 3600 * ((*lon_ptr) - (*wlon_ptr));

        // Compute new_hdg using the differences in lat/long
        *new_hdg_ptr = atan2(*lon_diff_ptr, *lat_diff_ptr)*180/PI;

        // Convert quadrant III/IV degrees to 180-360
        if (*new_hdg_ptr < 0.0)
                *new_hdg_ptr += 360.0;

        Plant_Compensator(*rng_ptr, *new_hdg_ptr, *cur_hdg_ptr);

        return 0;

} // end Navigate

/***************************************************************************
Function:       Plant_Compensator
```

```
Description:     PID control for MONTe.  P & D control turning rate.  I
        maintains forward speed.  Checks to keep wheels moving forward so no
        tank turns happen.  Also, limits maximum turn rate to avoid spinning.

Parameter:       1) Range to waypoint
                 2) Desired heading to waypoint
                 3) Current heading of MONTe

Return Value:    None
***************************************************************************/
int Plant_Compensator(double range, float desired_hdg, float current_hdg)
{
        float hdg_error;
        unsigned char k_p_left = 0;
        unsigned char k_d_left = 0;
        unsigned char k_p_right = 0;
        unsigned char k_d_right = 0;
        unsigned char k_i = 0;

        hdg_error = desired_hdg - current_hdg;

        if (hdg_error > 180.0)
                hdg_error -= 360.0;
         else if (hdg_error <= -180)
                hdg_error += 360;
/*      Proportional & Derivative control with gain scheduling  */
/*      Max turn if outside of +\- 80 degrees                   */
        if((hdg_error < 80.0) || (hdg_error > 80.0))
        {
                k_p_left = -MAX_TURN_DIFF * (char) hdg_error / (char) fabs(hdg_error);
                k_p_right = MAX_TURN_DIFF * (char) hdg_error / (char) fabs(hdg_error);
        }       else
                {       // Proportional gains for turning, in form of k_p = A*x^3
                        k_p_left = (char) (-K_P_coefficient * pow(hdg_error, 3.0));
                        k_p_right = (char) (K_P_coefficient * pow(hdg_error, 3.0));

                        // Derivative gain for turning
                }

/*      Integral control for maintaining velocity       */

/*      Assign speeds and provide limiting              */
        left_command = FWD_SPEED + L_CORRECT + k_p_left + k_i + k_d_left;
        right_command = FWD_SPEED + R_CORRECT + k_p_right + k_i + k_d_right;

/*      Keep commands inside design limitations incase of data corruption       */
        if(left_command > 117.0)        // Design speed range for left side
                left_command = 117.0;   // 65-117 (64 stop, 127 full)
                else if (left_command < 65.0)
                        left_command = 65.0;
```

```
        if(right_command > 246.0)        // Design speed range for right side
                right_command = 246.0;   // 191-246 (190 stop, 256 full)
                else if (right_command < 191.0)
                        right_command = 191.0;

        return 0;
} // end Plant_Control_Func
```

C.2 Waypoint Processing Node

Following code processes waypoint routes for MONTe.

```
/***************************************************************************
Title:      MONTe_Waypoint_Processing 0.0 (ROS Node)

Author:     Jason Hickle

Purpose:  Receives, store and publishes waypoint data for use in autonomous
        navigation.
        Handles the following functions:
                1) Monitor Command_Flags to determine behavior.
                2) Receive waypoints from New_Waypoint and store them.
                3) Publish current waypoint for use in Navigation.

Use:        Interface between user and navigation node.  When CF.route equals
        "true," program proceeds to receive and process a waypoint router.
        Can work with any other node that publishes to "New_Waypoints_T."
        Receives waypoints (up to 10) and stores for sending waypoints to the
        Navigation node. Will receive waypoints from Waypoint_Control (user
        input over VNC), or via Comms (wireless comms from base station)

          Waits for nav_mode 'N' to send new waypoint to navigation mode.
        Receives from Command_Flags.

          Node set up to allow future capability. Possible functions could
        be to set up search types, additional actions to perform upon reaching
        waypoint, etc.

          Loop rate set at 4 Hz.  This synchs with system loop rate.

ROS Notes:
        Name-               "MONTe_Waypoint_Processing"

        Publications-       "Current_Waypoint_T"
                            "Command_Flags_T"

        Subscriptions-      "New_Waypoints_T"
                            "Command_Flags_T"

        Messages-           Waypoint.msg
                            Command_Flags.msg
```

```
        Services—        None

Version History:

        —— Version 0.0 ——
        May 22nd, 2011
        LT Jason Hickle

          Establishes basic functionallity. Receives, stores and sends
        waypoints based off of Command_Flags_T

Notes:  Further information can be found on ROS Wiki page:
                http://www.ros.org/wiki/
*************************************************************************/

/*      Libraries               */
#include <ros/ros.h>

#include "MONTe/Waypoint.h"             // Message format (.msg)
#include "MONTe/Command_Flags.h"        // Message format (.msg)

/*      Defines                 */
// Debugging options, uncomment to enable
#define WP_INCOMING     // Publish info to ROS for incoming waypoints
#define WP_SEND         // Alert ROS when a new waypoint is sent

/*      CF holds all command flags. Expand as necessary.  Ensure publishing    /
/       utilizes default values for any flag unchanged to ensure flags are     /
/       not being changed unecessarily.                                        */
typedef struct
        {
                bool autonav;
                char mode;
                bool route;
        }CF;                            //Define flags structure

/*      WP_P stores all data necessary to navigate to, and determine           /
/       behavior mode for a route.                                             */
typedef struct
        {
                int num;
                double lat;
                double lon;
                char  action;
        }WP_P;                          //Define waypoint structure


/*      Global Variables        */
WP_P New_WP, waypoints[10];
CF Cmd_Flags;
```

```
int route_points = 0;   //  Stores length of route

/*      Functional Prototypes    */

void Command_FlagsCallback(const MONTe::Command_FlagsConstPtr& flags);

void WaypointCallback(const MONTe::WaypointConstPtr& way_pt);

int main(int argc, char **argv)
{
/*      ROS Initializations      */
        ros::init(argc, argv, "MONTe_Waypoint_Processing");  // Set up ROS node

        ros::NodeHandle n;  // set up handle for this node

            // Set up all publishers for node
        ros::Publisher nxt_wp_pub = n.advertise<MONTe::Waypoint>("Current_Waypoint_T", 1);
        ros::Publisher cmd_flg_pub = n.advertise<MONTe::Command_Flags>("Command_Flags_T", 1);

            // Set up all subscriptions for node
        ros::Subscriber sub_cf = n.subscribe("Command_Flags_T", 1, Command_FlagsCallback);
        ros::Subscriber sub_wp = n.subscribe("New_Waypoint_T", 1, WaypointCallback);

        ros::Rate loop_rate(4);             // Sets 4hz cycle for main loop

            // Set up message handles for communicating with topics.
        MONTe::Command_Flags flags, pub_flags;    // subscribe and publiser handles
        MONTe::Waypoint new_way_pt, next_way_pt; // subscribe and publiser handles

       // int total_wp_counter = 0;
       // int wp_entry_counter = 0;
        int current_wp_number = 0;

    pub_flags.auto_nav = 0;   // Initialize system in manual control, enroute to
    pub_flags.nav_mode = 'A'; // next waypoint, and no new route
    pub_flags.incoming_route = 0;
    cmd_flg_pub.publish(pub_flags);

    while(ros::ok())
    {
        ros::spinOnce();        // Callback to all active topics

        // Check for new route and fill in waypoints
        if(Cmd_Flags.route == 1)
        {       // Check for errors and fill in new waypoint
                if((New_WP.num >= 0) && (New_WP.num < 10))
                {       // Populate the waypoint queue
                        waypoints[New_WP.num].num = New_WP.num;
                        waypoints[New_WP.num].lat = New_WP.lat;
                        waypoints[New_WP.num].lon = New_WP.lon;
                        waypoints[New_WP.num].action = New_WP.action;
```

```
                                #ifdef WP_INCOMING
                                ROS_INFO("New WP: #%d, Lat- %lf, Lon- %lf, Action- %c,
Route Size- %d", waypoints[New_WP.num].num, waypoints[New_WP.num].lat,
waypoints[New_WP.num].lon, waypoints[New_WP.num].action, route_points);
                                #endif
                        }
// End of route received, place in "no new route"
                        if(New_WP.num == (route_points - 1))
                        {
                                pub_flags.incoming_route = 0;
                                pub_flags.auto_nav = Cmd_Flags.autonav;
                                pub_flags.nav_mode = Cmd_Flags.mode;
                                cmd_flg_pub.publish(pub_flags);
                        }
                } // End New Route if

                // Send waypoint if auto nav and next waypoint
                // check for auto_nav and if new waypoint is needed
                if((Cmd_Flags.autonav == 1) && (Cmd_Flags.mode == 'N'))
                {// New waypoint is available
                        if((current_wp_number < route_points) && (current_wp_number >= 0))
                        {  // Send next waypoint
                                next_way_pt.latitude = waypoints[current_wp_number].lat;
                                next_way_pt.longitude = waypoints[current_wp_number].lon;
                                next_way_pt.action = waypoints[current_wp_number].action;

                                nxt_wp_pub.publish(next_way_pt);

                                #ifdef WP_SEND
                                ROS_INFO("Waypoint %d sent to Nav.", current_wp_number);
                                #endif

                                current_wp_number++;

                        }       else  // Route complete, reset waypoints
                                {
                                        pub_flags.auto_nav = 0;
                                        pub_flags.nav_mode = 'A';
                                        pub_flags.incoming_route = Cmd_Flags.route;
                                        cmd_flg_pub.publish(pub_flags);

                                        #ifdef WP_SEND
                                        ROS_INFO("Route complete, in Manual Control.");
                                        #endif

                                        current_wp_number = 0;
                                } // End Send wp/route complete
                }

                loop_rate.sleep();      // Sleeps to maintain loop_rate
        } // End main while
```

```
} //end main

/*****************************************************************************
Function:          Command_FlagsCallback

Description:       Pulls waypoint from the Command_Flags_T topic

Parameter:         1) Pointer to ROS message from topic Command_Flags_T

Return Value:      None
*****************************************************************************/
void Command_FlagsCallback(const MONTe::Command_FlagsConstPtr& flags)
{
  Cmd_Flags.autonav = flags->auto_nav;
  Cmd_Flags.mode = flags->nav_mode;
  Cmd_Flags.route = flags->incoming_route;
} // end callback

/*****************************************************************************
Function:          WaypointCallback

Description:       Pulls waypoint from the Current_Waypoint_T topic

Parameter:         1) Pointer to ROS message from topic Current_Waypoint_T

Return Value:      None
*****************************************************************************/
void WaypointCallback(const MONTe::WaypointConstPtr& new_way_pt)
{
  New_WP.lat = new_way_pt->latitude;
  New_WP.lon = new_way_pt->longitude;
  New_WP.action = new_way_pt->action;
  New_WP.num = new_way_pt->wp_num;
  route_points = new_way_pt->route;
} // end callback
```

C.3 Monkey Driver

Following driver interfaces with Monkey board.

```
/*****************************************************************************
Title:       MONTe_Monkey_Driver 0.0 (ROS Node)

Author:      Jason Hickle

Purpose:  Node that allows MONTe to interface with CHIMU AHRS ("Monkey").
         Handles the following functions:
                   1) Read in serial data from Monkey AHRS.
                   2) Parse data string.
                   3) Publish data to Nav_Data topic.
```

```
Use:       Driver for Monkey nav unit.
        Output from Monkey needs to be the following comma separated values (CSV):
                $RM
                Satellites Tracked
                Fix Quality
                Latitude
                Longitude
                Heading
                Altitude
                Velocity (N/S)
                Velocity (E/W)
                Velocity (D/U)

          Additional parameters can be included by expanding the parsing function.

          Runs at loop rate of 4 Hz.  This is MONTe's program system loop
        rate.

ROS Notes:
        Name-               "MONTe_Monkey"

        Publications-       "Command_Flags_T"
                            "Nav_Data_T"

        Subscriptions-      None

        Messages-           Command_Flags.msg
                            Nav_Data.msg

        Services-           None

Version History:

        ---- Version 0.0 ----
        Mar 21st, 2011
        LT Jason Hickle

         Established link with Monkey at 115200 Baud. Code in place to
        receive full data string (waiting on adjustment of code on Monkey
        board itself). Successfully transmit nav data to "Nav_Data_T."

Notes:  Further information can be found on ROS Wiki page:
                http://www.ros.org/wiki/
****************************************************************************/

/*      Libraries               */
#include <ros/ros.h>


#include "MONTe/Nav_Data.h"             // Message format (.msg)
#include "MONTe/Command_Flags.h"        // Message format (.msg)
#include "MONTe_USB_Serial_Lib.h"
```

```
/*      Defines                 */
// Debugging options, uncomment to enable
//#define NAV_PRINT             // Print out nav data and check for successful
                                // parsing of command string

/*      NMEA_DATA is the data structure to hold navigational data received      /
/       from the Monkey/AHRS. Listed in order of data receieved via command     /
/       string. Use this to expand MONTe's capabilities for processing data.    */
        typedef struct {
                unsigned char command_str[3];
                int sat_track;
                int fix_quality;
                double latitude;
                double longitude;
                double heading;
                int altitude;
                double vel_N;
                double vel_E;
                double vel_D;
                } NMEA_DATA;

/*      Global Variables        */
NMEA_DATA GPS_Buffer;
char nav_buffer[254];                   // Buffer for nav data incoming from Monkey

/*      Functional Prototypes   */

int Parse_Monkey_Data();

void Print_Nav_Data();

int main(int argc, char **argv)
{
        int usb_fd;     // File descriptor for serial comms, included for future use
                        // for error checking

/*      ROS Initializations     */
            ros::init(argc, argv, "MONTe_Monkey");  // Set up ROS node

            ros::NodeHandle n;  // set up handle for this node

                // Set up all publishers for node
            ros::Publisher nav_pub = n.advertise<MONTe::Nav_Data>("Nav_Data_T", 1);
            ros::Publisher cmd_flg_pub = n.advertise<MONTe::Command_Flags>("Command_Flags_T", 1);

            ros::Rate loop_rate(4);             // Sets 4hz cycle for main loop

            MONTe::Nav_Data n_data;
            MONTe::Command_Flags flags;
/*      End ROS Initializations */
```

```
        usb_fd = OpenUSBSerialPort();

        while(ros::ok())
        {
                // Receive Navigation data
                // Update Command_Flags with fix quality (future implementation)
                if (ReadfromUSBSerialPort(nav_buffer, 254) > 0)
                {
#ifdef NAV_PRINT
                printf("\nStep 1");
#endif
                // Parse string data
                Parse_Monkey_Data();

#ifdef NAV_PRINT
                Print_Nav_Data();
#endif

#ifdef NAV_PRINT
                printf("\nStep 10\n");
#endif

/*      Update navigation data for publishing   */
                n_data.latitude = GPS_Buffer.latitude;
                n_data.longitude = GPS_Buffer.longitude;
                //n_data.heading = GPS_Buffer.heading;

/*      Publish navigation data to topic        */
                nav_pub.publish(n_data);

                } // end if

                loop_rate.sleep();      // Sleeps to maintain loop_rate
        } // end while loop

        CloseUSBSerialPort();
        return 0;
} //end main

/***************************************************************************
Function:       Parse_Monkey_Data

Description:    Tokenizes buffer to populate navigation data.  Add or subtract
        steps to adjust what gets pulled from string.

Parameter:      None

Return Value:   0- Success
***************************************************************************/
int Parse_Monkey_Data()
{
        static char *buf_ptr;
```

```
        buf_ptr = &nav_buffer[4];   // Start pointer at 4th element (satillites tracked)
#ifdef NAV_PRINT
                printf("\nStep 2");
#endif

#ifdef NAV_PRINT
                printf("\nStep 3");
#endif

#ifdef NAV_PRINT
                printf("\nStep 4");
#endif

        GPS_Buffer.sat_track = atoi(strtok(buf_ptr, ","));

#ifdef NAV_PRINT
                printf("\nStep 5");
#endif
        //GPS_Buffer.fix_quality = atoi(strtok(NULL, ","));
        //  Converts ascii fix quality to an integer

        GPS_Buffer.latitude = atof(strtok(NULL, ",")) / 10000;
        //  Converts ascii latitude to decimal degrees

#ifdef NAV_PRINT
                printf("\nStep 6");
#endif

        GPS_Buffer.longitude = atof(strtok(NULL, ",")) / 10000;
        //  Converts ascii longitude to decimal degrees

        //GPS_Buffer.heading = atof(strtok(NULL, ",")) / 10;
        //  Converts ascii heading to double with 2 decimal places

        GPS_Buffer.altitude = atoi(strtok(NULL, ","));
        //  Converts ascii altitude to integer meters (MSL)

#ifdef NAV_PRINT
                printf("\nStep 7");
#endif

        GPS_Buffer.vel_N = atof(strtok(NULL, ",")) / 10;
        //  Converts ascii N/S velocity to double with 2 decimal places

#ifdef NAV_PRINT
                printf("\nStep 8");
#endif

        GPS_Buffer.vel_E = atof(strtok(NULL, ",")) / 10;
        //  Converts ascii E/W velocity to double with 2 decimal places
```

```
#ifdef NAV_PRINT
                printf("\nStep 9");
#endif

        GPS_Buffer.vel_D = atof(strtok(NULL, ",")) / 10;
        // Converts ascii U/D velocity to double with 2 decimal places

        buf_ptr = NULL;

        return 0;

} // end callback

/***************************************************************************
Function:       Print_Nav_Data

Description:    Printing function for troubleshooting. Uncomment/add lines
        to print out additional data.

Parameter:      None

Return Value:   None
***************************************************************************/
void Print_Nav_Data()
{
        printf("\n\tCommand string\t%s\n", GPS_Buffer.command_str);
        printf("Satellites tracked\t%d\n", GPS_Buffer.sat_track);
        //printf("\n\tFix quality\t%s\n", GPS_Buffer.fix_quality);
        printf("\tCurrent Latitude\t%f\n", GPS_Buffer.latitude);
        printf("\tCurrent Longitude\t%f\n", GPS_Buffer.longitude);
        //printf("\tCommand String\t%f\n", GPS_Buffer.heading);
        printf("\tCommand String\t%d\n", GPS_Buffer.altitude);
        printf("\tCommand String\t%f\n", GPS_Buffer.vel_N);
        printf("\tCommand String\t%f\n", GPS_Buffer.vel_E);
        printf("\tCommand String\t%f\n", GPS_Buffer.vel_D);

} // end callback
```

C.4 USB-Serial Library

Library for running RS-232 serial communications over USB port. Used in conjunction with MONTe_Monkey_Driver.cpp.

```
/***************************************************************************
Title:      MONTe_USB_Serial_Lib 0.0

Author:     Jason Hickle

Purpose:  List of functions to read and write serial data over a USB-Serial
        for use with MONTe on Linux.
```

```
Use:        Include "MONTe_USB_Serial_Lib.h" in the header of each node needed.

            Include source file as part of CMakelist.txt as follows:
rosbuild_add_executable(node_name src/node_name.cpp src/MONTe_USB_Serial_Lib.cpp)

        Verify that serial handle (/dev/ttyUSB0) is correct for system.

Version History:

        —— Version 0.0 ——
        Mar 21st, 2011
        LT Jason Hickle

        Open and Close USB/Serial port.  Reads data from serial port.
*************************************************************************/

/*      Libraries               */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include <errno.h>
#include "MONTe_USB_Serial_Lib.h"

/*      Defines                 */

/*      Global Variables        */
static int fd = 0;
static struct termios oldtio;

/*      Functions               */

/*************************************************************************
Function:       OpenUSBSerialPort

Description:    Takes port number and opens appropriate serial connection.
        Will save old port data

Parameter:      1) sPortNumber - pointer to comm port ttyS(X), ie 0 for ttyS0
                etc.

Return Value:   fd - file descriptor for port
*************************************************************************/
int OpenUSBSerialPort()
{
    char sPortName[64] = "/dev/ttyUSB0";  // Hardcoded until generic method works

    // make sure port is closed
```

```
    CloseUSBSerialPort();

    fd = open(sPortName, O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd < 0)
    {
        printf("open error %d %s\n", errno, strerror(errno));
    }
    else
    {
        struct termios my_termios;
        tcgetattr(fd, &my_termios);

        oldtio = my_termios;  // Save port attributes to restore later

        tcflush(fd, TCIFLUSH);

        my_termios.c_cflag = B115200 | CS8 | CREAD | CLOCAL | HUPCL;

        cfsetospeed(&my_termios, B115200);
        tcsetattr(fd, TCSANOW, &my_termios);
    } // end if

    return fd;
} // end K_OpenSerialPort

/*************************************************************************
Function:           CloseUSBSerialPort

Description:        Checks to see if port is open and then closes it.  Returns
        port attributes to original configuration.

Parameter:          None

Return Value:       None
*************************************************************************/
void CloseUSBSerialPort()
{
        // you may want to restore the saved port attributes
    if (fd > 0)
    {
        tcsetattr(fd, TCSANOW, &oldtio);
        close(fd);
    } // end if
} // end K_CloseSerialPort

/*************************************************************************
Function:           WritetoUSBSerialPort

Description:

Parameter:          1)
```

```
Return Value:
*************************************************************************/

int WritetoUSBSerialPort(char* psOutput)
{
    int iOut;
    if (fd < 1) // Port is not open, return -1
    {
        return -1;
    } // end if

    iOut = write(fd, psOutput, 1);      // Set to 1 so only one byte is xmitted

    if (iOut < 0)
    {   // Place in ROS_INFO statement!!!!!!!
        //printf("write error %d %s\n", errno, strerror(errno));
    }

    return iOut;
} // end K_WritetoSerialPort

/*************************************************************************
Function: ReadfromUSBSerialPort

Description:

Parameter:      1)

Return Value:
*************************************************************************/

int ReadfromUSBSerialPort(char* psResponse, int iMax)
{
    int iIn;
    //printf("in ReadAdrPort iMax=%d\n", iMax);

    if (fd < 1)
    {
        printf(" port is not open\n");
        return -1;
    } // end if

    strncpy (psResponse, "N/A", iMax<4?iMax:4);
    iIn = read(fd, psResponse, iMax-1);

    if (iIn < 0)
    {
        if (errno == EAGAIN)
        {
                return 0; // assume that command generated no response
        }
        else
```

```
        {
                printf("read error %d %s\n", errno, strerror(errno));
        } // end if
    }
    else
    {
        psResponse[iIn<iMax?iIn:iMax] = '\0';
            // printf("read %d chars: %s\n", iIn, psResponse);
    } // end if

    return iIn;
} // end ReadAdrPort
```

```
/***************************************************************************
Title:      MONTe_USB_Serial_Lib 0.0 (ROS Node)

Author:     Jason Hickle

Purpose:    Header file for MONTe serial comms.
***************************************************************************/

/*        Libraries                  */
#include <ros/ros.h>

int OpenUSBSerialPort();
void CloseUSBSerialPort();
int WritetoUSBSerialPort(char* psOutput);
int ReadfromUSBSerialPort(char* psResponse, int iMax);
```

C.5 Plant Control Driver

Following driver interfaces with Sabertooth 2x12 Motor Drivers.

```
/***************************************************************************
Title:      MONTe_Plant_Control 0.0.4.2 (ROS Node)

Author:     Jason Hickle

Purpose:    Driver node that allows MONTe to control the whegs and water jets.
        Handles the following functions:
                1) Manual Control
                2) Communicate with Sabertooth2x12 motor drivers

Use:        Receives command data from the command topics and executes.
        Commands sent to Sabertooth2x12 motor driver via RS-232 interface,
        over port /dev/ttyS0. Can run both whegs and waterjets over same
        interface as long as commands have correct address in packetized
        serial mode.

            For plant control running in Simplified Serial mode for Sabertooth
```

```
        2x12 motor drivers: Each plant command will send 4 bytes of data.
                1st byte: Left motor command (1-full rev, 64-stop, 127-full fwd).
                2nd Byte: Left motor command (128-full rev, 190-stop, 256-full fwd).


          For plant control running in Packetized Serial mode for Sabertooth
        2x12 motor drivers:  Each plant command will send 4 bytes of data.
                1st byte: motor controler address.
                2nd Byte: command (fwd, rev, left turn, right turn)
                3rd Byte: speed (0-127)
                4th Byte: checksum, (address+command+speed) & 0b01111111

          Manual control receives desired speed and turn rate from topic, and
        parses data.  Converts signed int8_t to unsigned char for xmit to
        motor controllers.

          Runs at system loop rate of 4 Hz.

ROS Notes:
        Name-               "MONTe_Plant_Control"

        Publications-    None

        Subsrciptions-   "Plant_Commands_T"

        Messages-          Plant_Command.msg

        Services-          None

Version History:

        ---- Version 0.0.4.2 ----
        Apr 26th, 2011
        LT Jason Hickle

        - Enable simplified serial control for Sabertooth2x12. Packetized serial is problematic.

        ---- Version 0.0.4.1 ----
        Apr 26th, 2011
        LT Jason Hickle

        - Enable packetized serial control for Sabertooth2x12.  Using alternate logic
        for parsing data commands.  Instituted a 1hz spin rate for sending commands.
        Using new message (v2) for sending command data.  Moderate success, sends right
        commands for a command or two, then abberent behavior occurs.

        ---- Version 0.0.4 ----
        Apr 26th, 2011
        LT Jason Hickle

        - Enable packetized serial control for Sabertooth2x12.  Non-deterministic behavior.
```

```
        —— Version 0.0.3 ——
        Apr 26th, 2011
        LT Jason Hickle

        – Enabled simplified serial control for Sabertooth2x12. Successful

        —— Version 0.0.2 ——
        Mar 31st, 2011
        LT Jason Hickle

        – Successfully receives commands from control topic, and correctly
        parses them.  Unable to get packetized serial to work with Sabertooth2x12
        motor controllers.  Will persue simplified serial and come back to
        packetized serial later.

        —— Version 0.0.1 ——
        Mar 31st, 2011
        LT Jason Hickle

        – Successfully receives commands from control topic, and correctly
        parses them.

        —— Version 0.0 ——
        Mar 21st, 2011
        LT Jason Hickle

        Enable Manual Control mode in main control loop.

Notes:  Further information can be found on ROS Wiki page:
                http://www.ros.org/wiki/
***************************************************************************/

/*      Libraries               */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include <errno.h>
#include <stdint.h>
#include <unistd.h>

#include <ros/ros.h>
#include "std_msgs/String.h"
#include "MONTe/Plant_Command.h"        // Message format (.msg)
#include "MONTe_Serial_Lib.h"
#include <sstream>

/*      Debugging               */
```

```
// Debugging options, uncomment to enable
#define MOTOR_CONTROL_DEBUG          // Prints out commands from manual cmd topic
//#define FLOW_CHECK                 // Prints out stages of code, uncomment to check that
                                     // individual sections are running

/*      Defines                 */
const char MOTOR_PORT_NUMBER = 0;    // For adding ttyS(X) selection later

/*      Global Variables        */

// Control Variables
uint8_t MONTe_Left, MONTe_Right;     // Received data from Manual_Commands_T
uint8_t temp_left = 0;               // temporary storage values to check for
uint8_t temp_right = 0;              // if manual commands have changed


/*      Functional Prototypes   */
void Plant_CommandCallback(const MONTe::Plant_CommandConstPtr& command);

int Manual_Control_Simplified_Parser();

int Plant_Control_Simplified(unsigned char* s_speed, const char* port_num);

int main(int argc, char **argv)
{
/*      Variables       */

        // Flags
        bool Manual_Command_F = 1;
        //bool New_Command_F = 1;

/*      Initializations         */
        // Perform initializations for ROS
           ros::init(argc, argv, "MONTe_Plant_Control");  // Set up ROS node

           ros::NodeHandle n;  // set up handle for this node

                // Set up all subscriptions for node
           ros::Subscriber man_cmd = n.subscribe("Plant_Command_T", 100, Plant_CommandCallback);

           ros::Rate loop_rate(4);              // Sets 4hz cycle for main loop

           MONTe::Plant_Command command;

        // Initialize variables
        MONTe_Left = 0;
        MONTe_Right = 0;
        char portno = (char) MOTOR_PORT_NUMBER;
        K_OpenSerialPort(&portno);

/*      Main control loop.                                                  *
*               Performs the following: - Poll all topics                   *
```

```
*                                       - Manual control                    *
*                                       - Maintains desired loop rate      */

        while(ros::ok())
        {
        ros::spinOnce();         // Callback to all active topics

#ifdef MOTOR_CONTROL_DEBUG
        ROS_INFO("Topic data: Left Speed %d\tRight Speed %d", MONTe_Left, MONTe_Right);
#endif

        if(Manual_Command_F)
                {
                        Manual_Control_Simplified_Parser();
                }

        loop_rate.sleep();       // Sleeps to maintain loop_rate

        } // end while loop

        K_CloseSerialPort();
} //end main

/*************************************************************************
Function:       Plant_CommandCallback

Description:    Pulls manual commands from the Plant_Command topic

Parameter:      1) Pointer to ROS message from topic Plant_Command

Return Value:   None
*************************************************************************/
void Plant_CommandCallback(const MONTe::Plant_CommandConstPtr& command)
{
  MONTe_Left = command->left;
  MONTe_Right = command->right;
} // end callback

/*************************************************************************
Function:       Manual_Control_Simplified_Parser

Description:    Pulls manual control data for forward/reverse speed and
        turning rate.  Calls Plant_Control to send commands to Sabertooth
        2x12 motor drivers.

Parameter:      Void

Return Value:   0       No new command
                1       New command sent
*************************************************************************/

int Manual_Control_Simplified_Parser()
```

```
{

unsigned char sabertooth_left_speed;    // Desired left speed.
unsigned char sabertooth_right_speed;   // Desired right speed

/*      Check if new command is received. Return 0 if not.      */
        if((MONTe_Left == temp_left) && (MONTe_Right == temp_right))
        {
                return 0;
        }

#ifdef MOTOR_CONTROL_DEBUG
        ROS_INFO("Current: Left-%d, Right-%d\nOrdered: Left-%d, Right-%d.",
                        temp_left, temp_right, MONTe_Left, MONTe_Right);
#endif

        sabertooth_left_speed = (unsigned char) MONTe_Left;
        sabertooth_right_speed = (unsigned char) MONTe_Right;

/*      Send command to left motor.     */
        Plant_Control_Simplified(&sabertooth_left_speed, &MOTOR_PORT_NUMBER);

/*      Send command to right motor.    */
        Plant_Control_Simplified(&sabertooth_right_speed, &MOTOR_PORT_NUMBER);

        temp_left = MONTe_Left;
        temp_right = MONTe_Right;

        return 1;
} // end Manual_Command_Parser

/***********************************************************************
Function:       Plant_Control

Description:     Takes a command string  and transmits to Sabertooth2x12
                motor drivers via RS-232. Opens port, writes address/command
                /data/checksum, then closes port. This is for simplified
                serial mode.

Parameter:      1) Pointer to speed command
                2) Port number of serial eg 0 of "ttyS0"

Return Value:   0       Command sent successfully
                -1      Port failed to open
                -2      Write failed
***********************************************************************/
int Plant_Control_Simplified(unsigned char* s_speed, const char* port_num)
{
        char port;
        port = (char) *port_num;

        //if(K_OpenSerialPort(&port)<0)
```

```
                //return -1;

        if(K_WritetoSerialPort(s_speed)<0)
                return -2;

        //K_CloseSerialPort();

        return 0;
}
```

C.6 Serial Library

Library for running RS-232 serial communications over USB port. Used in conjunction with MONTe_Plant_Control.cpp.

```
/***************************************************************************
Title:     MONTe_Serial_Lib 0.0

Author:    Jason Hickle

Purpose:   List of functions to write serial data for use with MONTe on Linux

Use:       Include "MONTe_Serial_Lib.h" in the header of each node needed.

           Include source file as part of CMakelist.txt as follows:
rosbuild_add_executable(node_name src/node_name.cpp src/MONTe_Serial_Lib.cpp)

        Verify that serial handle (/dev/ttyS0) is correct for system.

Version History:

        --- Version 0.0 ---
        Mar 21st, 2011
        LT Jason Hickle

        Open and Close Serial port.  Write data to serial port.
***************************************************************************/

/*      Libraries               */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include <errno.h>
#include "MONTe_Serial_Lib.h"

/*      Defines                 */
```

```
/*      Global Variables        */
static int fd = 0;
static struct termios oldtio;

/*      Functions               */

/***********************************************************************
Function:       K_OpenSerialPort

Description:    Takes port number and opens appropriate serial connection.
        Will save old port data

Parameter:      1) sPortNumber - pointer to comm port ttyS(X), ie 0 for ttyS0
                etc.

Return Value:   fd - file descriptor for port
***********************************************************************/
int K_OpenSerialPort(char* sPortNumber)
{
    char sPortName[64] = "/dev/ttyS0";  // Hardcoded until generic method works
    //sprintf(sPortName, "/dev/ttyS%c", *sPortNumber); // Not working, need to research

    // make sure port is closed
    K_CloseSerialPort();

    fd = open(sPortName, O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd < 0)
    {
        printf("open error %d %s\n", errno, strerror(errno));
    }
    else
    {
        struct termios my_termios;
        tcgetattr(fd, &my_termios);

        oldtio = my_termios;  // Save port attributes to restore later

        tcflush(fd, TCIFLUSH);

        my_termios.c_cflag = B9600 | CS8 | CREAD | CLOCAL | HUPCL;

        cfsetospeed(&my_termios, B9600);
        tcsetattr(fd, TCSANOW, &my_termios);
    } // end if

    return fd;
} // end K_OpenSerialPort

/***********************************************************************
Function:       K_CloseSerialPort
```

```
Description:     Checks to see if port is open and then closes it.  Returns
        port attributes to original configuration.

Parameter:       None

Return Value:    None
*************************************************************************/
void K_CloseSerialPort()
{
        // you may want to restore the saved port attributes
    if (fd > 0)
    {
        tcsetattr(fd, TCSANOW, &oldtio);
        close(fd);
    } // end if
} // end K_CloseSerialPort

/*************************************************************************
Function:        K_WritetoSerialPort

Description:

Parameter:       1)

Return Value:
*************************************************************************/

int K_WritetoSerialPort(unsigned char* psOutput)
{
    int iOut;
    if (fd < 1) // Port is not open, return -1
    {
        return -1;
    } // end if

    iOut = write(fd, psOutput, 1);       // Set to 1 so only one byte is xmitted

    if (iOut < 0)
    {   // Place in ROS_INFO statement!!!!!!!
        //printf("write error %d %s\n", errno, strerror(errno));
    }

    return iOut;
} // end K_WritetoSerialPort

/*************************************************************************
Function: K_ReadfromSerialPort

Description:

Parameter:       1)
```

```
Return Value:
************************************************************************/

/*int K_ReadfromSerialPort(int8_t* psResponse, int iMax)
{
    int iIn;
    printf("in ReadAdrPort iMax=%d\n", iMax);
    if (fd < 1)
    {
        printf(" port is not open\n");
        return -1;
    } // end if
    strncpy (psResponse, "N/A", iMax<4?iMax:4);
    iIn = read(fd, psResponse, iMax-1);
    if (iIn < 0)
    {
        if (errno == EAGAIN)
        {
                return 0; // assume that command generated no response
        }
        else
        {
                printf("read error %d %s\n", errno, strerror(errno));
        } // end if
    }
    else
    {
        psResponse[iIn<iMax?iIn:iMax] = '\0';
            printf("read %d chars: %s\n", iIn, psResponse);
    } // end if

    return iIn;
} // end ReadAdrPort
*/
```

```
/************************************************************************
Title:     MONTe_Serial_Lib 0.0

Author:    Jason Hickle

Purpose:   List of functions to write serial data for use with MONTe on Linux

Use:       Include "MONTe_Serial_Lib.h" in the header of each node needed.

           Include source file as part of CMakelist.txt as follows:
rosbuild_add_executable(node_name src/node_name.cpp src/MONTe_Serial_Lib.cpp)

        Verify that serial handle (/dev/ttyS0) is correct for system.

Version History:

        —— Version 0.0 ——
```

```
        Mar 21st, 2011
        LT Jason Hickle

        Open and Close Serial port.  Write data to serial port.
*************************************************************************/

/*      Libraries               */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include <errno.h>
#include "MONTe_Serial_Lib.h"

/*      Defines                 */

/*      Global Variables        */
static int fd = 0;
static struct termios oldtio;

/*      Functions               */

/*************************************************************************
Function:       K_OpenSerialPort

Description:    Takes port number and opens appropriate serial connection.
        Will save old port data

Parameter:      1) sPortNumber - pointer to comm port ttyS(X), ie 0 for ttyS0
                etc.

Return Value:   fd - file descriptor for port
*************************************************************************/
int K_OpenSerialPort(char* sPortNumber)
{
    char sPortName[64] = "/dev/ttyS0";  // Hardcoded until generic method works
    //sprintf(sPortName, "/dev/ttyS%c", *sPortNumber); // Not working, need to research

    // make sure port is closed
    K_CloseSerialPort();

    fd = open(sPortName, O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd < 0)
    {
        printf("open error %d %s\n", errno, strerror(errno));
    }
    else
    {
        struct termios my_termios;
```

```
        tcgetattr(fd, &my_termios);

        oldtio = my_termios;  // Save port attributes to restore later

        tcflush(fd, TCIFLUSH);

        my_termios.c_cflag = B9600 | CS8 | CREAD | CLOCAL | HUPCL;

        cfsetospeed(&my_termios, B9600);
        tcsetattr(fd, TCSANOW, &my_termios);
    } // end if

    return fd;
} // end K_OpenSerialPort

/*************************************************************************
Function:        K_CloseSerialPort

Description:     Checks to see if port is open and then closes it.  Returns
        port attributes to original configuration.

Parameter:       None

Return Value:    None
*************************************************************************/
void K_CloseSerialPort()
{
        // you may want to restore the saved port attributes
    if (fd > 0)
    {
        tcsetattr(fd, TCSANOW, &oldtio);
        close(fd);
    } // end if
} // end K_CloseSerialPort

/*************************************************************************
Function:        K_WritetoSerialPort

Description:

Parameter:       1)

Return Value:
*************************************************************************/

int K_WritetoSerialPort(unsigned char* psOutput)
{
    int iOut;
    if (fd < 1) // Port is not open, return -1
    {
        return -1;
    } // end if
```

```
    iOut = write(fd, psOutput, 1);        // Set to 1 so only one byte is xmitted

    if (iOut < 0)
    {   // Place in ROS_INFO statement!!!!!!!
        //printf("write error %d %s\n", errno, strerror(errno));
    }

    return iOut;
} // end K_WritetoSerialPort

/*************************************************************************
Function: K_ReadfromSerialPort

Description:

Parameter:        1)

Return Value:
*************************************************************************/

/*int K_ReadfromSerialPort(int8_t* psResponse, int iMax)
{
    int iIn;
    printf("in ReadAdrPort iMax=%d\n", iMax);
    if (fd < 1)
    {
        printf(" port is not open\n");
        return -1;
    } // end if
    strncpy (psResponse, "N/A", iMax<4?iMax:4);
    iIn = read(fd, psResponse, iMax-1);
    if (iIn < 0)
    {
        if (errno == EAGAIN)
        {
                return 0; // assume that command generated no response
        }
        else
        {
                printf("read error %d %s\n", errno, strerror(errno));
        } // end if
    }
    else
    {
        psResponse[iIn<iMax?iIn:iMax] = '\0';
            printf("read %d chars: %s\n", iIn, psResponse);
    } // end if

    return iIn;
} // end ReadAdrPort
*/
```

C.7 Keyboard Control Node

Following code performs manual control of MONTe.

```
/***********************************************************************
Title:     MONTe_Keyboard_Control 0.5 (ROS Node)

Author:    Jason Hickle

Purpose:  Node that allows MONTe to be controlled via keyboard over a VNC
        server.
        Handles the following functions:
                1) Take keyboard commands
                2) Change the speed and turning rates
                3) Publishes manual commands to ROS topic
                4) Updates command flags to manual control upon command

Use:       Use the arrow keys to move. P stops the robot. U prompts user to
        change speed.  I prompts user to change turn rate.  Works with
        simplified serial mode.

ROS Notes:
        Name-              "MONTe_Keyboard_Control"

        Publications-      "Plant_Commands_T"
                           "Command_Flags_T"

        Subscriptions-     None

        Messages-          Plant_Command.msg
                           Command_Flags.msg

        Services-          None

Version History:

        ---- Version 0.5 ----
        Apr 26th, 2011
        LT Jason Hickle

         Added publishing to "Command_Flags_T" to update when receiving manual commands.

        ---- Version 0.4 ----
        Apr 26th, 2011
        LT Jason Hickle

         Using simplified serial command for motors and utilizing Plant_Command
        msg format. Allow user to change fwd/rev speed and turning rate.
```

```
        —— Version 0.3 ——
        Apr 26th, 2011
        LT Jason Hickle

         Using packetized serial command for motors and utilizing Manual_Command_2
        msg format. Allow user to change speed and turning rate.

        —— Version 0.2 ——
        Apr 10th, 2011
        LT Jason Hickle

         Changed commands to simplified serial format from prior packetized
        serial format.  Sends new msg format to "Manual_Commands_T".  Return to
        Manual_Command.msg vice Manual_Command_Simplified.msg if running in
        packetized serial mode.

        —— Version 0.1 ——
        Mar 31st, 2011
        LT Jason Hickle

         Added debug options to verify correct commands.  Fixed incorrect keycodes.

        —— Version 0.0 ——
        Mar 31st, 2011
        LT Jason Hickle

         Enable keyboard commands to send manual control signals over to
        topic Manual_Commands_T for packetized serial mode.

Notes:  Further information can be found on ROS Wiki page:
                http://www.ros.org/wiki/

*************************************************************************/

/*      Libraries               */
#include <ros/ros.h>
#include <signal.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include "MONTe/Plant_Command.h"
#include "MONTe/Command_Flags.h"

/*      Debugging               */
#define COMMAND_VALUE_PRINT

/*      Defines                 */
#define KEYCODE_RIGHT     0x43
#define KEYCODE_LEFT      0x44
#define KEYCODE_UP        0x41
#define KEYCODE_DOWN      0x42
#define KEYCODE_Q         0x71
```

```
#define KEYCODE_SPACE    0x20
#define KEYCODE_U        0x75
#define KEYCODE_I        0x69
#define KEYCODE_Y        0x79
#define KEYCODE_J        0x6a
#define KEYCODE_H        0x68

const unsigned char STOP_R = 190;
const unsigned char STOP_L = 64;

/*        Global Variables          */

// Keyboard control variables
int kfd = 0;
struct termios cooked, raw;

// Command variables
unsigned char fwd_spd = 30;             // Value for going forward
unsigned char rev_spd = 20;             // Value for going reverse
unsigned char turn_spd = 10;            // Differential turning speed
unsigned char temp_fwd_spd = 30;        // temporary storage variables
unsigned char temp_rev_spd = 20;
unsigned char temp_turn_spd = 20;
unsigned char temp_l_turn = 64;         // Temp vars for computing turning rates.
unsigned char temp_r_turn = 190;
unsigned char temp_turn = 10;
unsigned char *fwd_spd_ptr, *rev_spd_ptr, *turn_spd_ptr, *temp_l_turn_ptr, *temp_r_turn_ptr;
                                        // Pointers for code optimization
unsigned char L_Correction = 3;                 // Calibration coefficients
unsigned char R_Correction = 1;
unsigned char temp_l_correct, temp_r_correct;
unsigned char *l_correct_ptr, *r_correct_ptr;
unsigned char turn_flag = 0;

/*        Functional Prototypes    */
void quit(int sig);

unsigned char turn_simplified(unsigned char speed, char flag);

int main(int argc, char **argv)
{
// Initializations
        // Perform initializations for ROS
        ros::init(argc, argv, "MONTe_Keyboard_Control");  // Set up ROS node

        ros::NodeHandle n;  // set up handle for this node

        // Set up all publications for node
        ros::Publisher man_cmd_pub = n.advertise<MONTe::Plant_Command>("Plant_Command_T", 1);
        ros::Publisher cmd_flg_pub = n.advertise<MONTe::Command_Flags>("Command_Flags_T", 1);

        MONTe::Plant_Command cmd;
```

```
        MONTe::Command_Flags flags;
        // End ROS initialization

/*      Variable and pointer initialization     */
        cmd.left = STOP_L;
        cmd.right = STOP_R;
        fwd_spd_ptr = &fwd_spd;
        rev_spd_ptr = &rev_spd;
        turn_spd_ptr = &turn_spd;
        temp_l_turn_ptr = &temp_l_turn;
        temp_r_turn_ptr = &temp_r_turn;
        l_correct_ptr = &L_Correction;
        r_correct_ptr = &R_Correction;

        //  Set the auto_nav flag to 0 (manual) when manual command is received
        flags.auto_nav = 0;

        signal(SIGINT,quit);

        // Initialize keyboard
                // get the console in raw mode
        tcgetattr(kfd, &cooked);
        memcpy(&raw, &cooked, sizeof(struct termios));
        raw.c_lflag &=~ (ICANON | ECHO);

                // Setting a new line, then end of file
        raw.c_cc[VEOL] = 1;
        raw.c_cc[VEOF] = 2;
        tcsetattr(kfd, TCSANOW, &raw);

        std::cout << "Reading from keyboard\n";
        std::cout << "---------------------------\n";
        std::cout << "Use arrow keys to move MONTe.\n";
        std::cout << "P stops MONTe.\n";
        std::cout << "Y to enter new reverse speed.\n";
        std::cout << "U to enter new reverse speed.\n";
        std::cout << "I to enter new turn rate.\n";
        std::cout << "H to enter new left wheg correction.\n";
        std::cout << "J to enter new right wheg correction.\n";
        std::cout << "Current settings: Fwd speed " << (int) fwd_spd << " Rev speed "
                << (int) rev_spd << " Turn rate " << (int) turn_spd << " \n";
        //puts("Q quits ROS.");

        while(ros::ok())
        {
                char c;
                bool dirty = false;

        // get the next event from the keyboard
                if(read(kfd, &c, 1) < 0)
                {
                        perror("read():");
```

```
        exit(-1);
    }

    ROS_DEBUG("value: 0x%02X\n", c);

    switch(c)
    {
        case KEYCODE_LEFT: // Turn left
            ROS_DEBUG("LEFT");
    // Computes turn differential speed for whegs and adds correction factor
            *temp_l_turn_ptr = turn_simplified(STOP_L + fwd_spd, 0);
            *temp_r_turn_ptr = turn_simplified(STOP_R + fwd_spd, 1);
            cmd.left = *temp_l_turn_ptr;
            cmd.right = *temp_r_turn_ptr;

            dirty = true;
            ROS_INFO("Going Left\n");

            break;

        case KEYCODE_RIGHT: // Turn right
            ROS_DEBUG("RIGHT");
    // Computes turn differential speed for whegs and adds correction factor
            *temp_l_turn_ptr = turn_simplified(STOP_L + fwd_spd, 1);
            *temp_r_turn_ptr = turn_simplified(STOP_R + fwd_spd, 0);
            cmd.left = *temp_l_turn_ptr;
            cmd.right = *temp_r_turn_ptr;

            dirty = true;
            ROS_INFO("Going Right\n");

            break;

        case KEYCODE_UP:        // Go forward
            ROS_DEBUG("UP");
            // Enters desired fwd speed and adds calibration correction
            cmd.left = STOP_L + *fwd_spd_ptr + *l_correct_ptr;
            cmd.right = STOP_R + *fwd_spd_ptr + *r_correct_ptr;

            dirty = true;
            ROS_INFO("Going Forward\n");

            break;

        case KEYCODE_DOWN:      // Go in reverse
            ROS_DEBUG("DOWN");
            // Enters desired rev speed and applies calibration correction
            cmd.left = STOP_L - *rev_spd_ptr - *l_correct_ptr;
            cmd.right = STOP_R - *rev_spd_ptr - *r_correct_ptr;

            dirty = true;
```

```
            ROS_INFO("Going Backwards\n");

            break;

    case KEYCODE_SPACE:        // Stop
            ROS_DEBUG("STOP");

            cmd.left = STOP_L;
            cmd.right = STOP_R;

            dirty = true;
            ROS_INFO("Stopping\n");

            break;

    case KEYCODE_Y: // Enter new forward speed
            ROS_DEBUG("Enter new forward speed");

            do      {       // User inputs speed and checks for valid input
                    std::cout << "\nEnter forward speed (0-50).";
                    scanf("%d", &temp_fwd_spd);
                    } while (temp_fwd_spd > 50);

            fwd_spd = temp_fwd_spd;

            dirty = true;
            ROS_INFO("New forward speed %d\n", fwd_spd);

            continue; // Return to top of while loop

    case KEYCODE_U: // Enter new reverse speed
            ROS_DEBUG("Enter new reverse speed");

            do      {       // User inputs speed and checks for valid input
                    printf("\nEnter new reverse speed (0-50).");
                    scanf("%d", &temp_rev_spd);
                    } while (temp_rev_spd > 50);

            rev_spd = temp_rev_spd;

            dirty = true;
            ROS_INFO("New reverse speed %d\n", rev_spd);

            continue; // Return to top of while loop

    case KEYCODE_I: // Enter new turn rate
            ROS_DEBUG("Enter turn rate");

            do      {       // User inputs speed and checks for valid input
                    printf("\nEnter turn rate (0-40).");
                    scanf("%d", &temp_turn_spd);
                    } while (temp_turn_spd > 40);
```

```
                                turn_spd = temp_turn_spd;

                                dirty = true;
                                ROS_INFO("New turn rate %d\n", turn_spd);

                                continue; // return to top of while loop

                        case KEYCODE_H: // Enter correction factor for left set of whegs
                                ROS_DEBUG("Enter left speed correction");

                                do      {       // User inputs speed and checks for valid input
                                        printf("\nEnter left correction factor (0-13).");
                                        scanf("%d", &temp_l_correct);
                                        } while (temp_l_correct > 13);

                                *l_correct_ptr = temp_l_correct;

                                dirty = true;
                                ROS_INFO("New left correction %d\n", *l_correct_ptr);

                                continue; // return to top of while loop

                        case KEYCODE_J: // Enter correction factor for right set of whegs
                                ROS_DEBUG("Enter left speed correction");

                                do      {// User inputs speed and checks for valid input
                                        printf("\nEnter right correction factor (0-13).");
                                        scanf("%d", &temp_r_correct);
                                        } while (temp_r_correct > 13);

                                *r_correct_ptr = temp_r_correct;

                                dirty = true;
                                ROS_INFO("New left correction %d\n", *r_correct_ptr);

                                continue; // return to top of while loop

                        default:
                                continue; // return to top of while loop
                } // end switch

#ifdef COMMAND_VALUE_PRINT
ROS_INFO("Speed: Left %d\tRight %d", cmd.left, cmd.right);
#endif

                        man_cmd_pub.publish(cmd);
                        cmd_flg_pub.publish(flags);

                        dirty=false;
        } // End main while loop
} //end main
```

```
/***************************************************************************
Function: quit

Description:

Parameter:      1)

Return Value:
***************************************************************************/
void quit(int sig)
{
  tcsetattr(kfd, TCSANOW, &cooked);
  ros::shutdown();
  exit(0);
}

/***************************************************************************
Function:       turn_simplified

Description:  Takes turn signals and outputs proper simplified serial value

Parameter:      1) Forward speed
                2) Inside whegs (0), outside whegs (1)

Return Value:   Simplified serial value (64-127 for left whegs, 190-255 for right
        whegs).
***************************************************************************/
unsigned char turn_simplified(unsigned char speed, char flag)
{
        unsigned char final_speed;

        if(speed < 64)
                return 0;       // Bad forward turn command, sends a 0 which will
                                // stop the motor

if(speed >= 190)
{       if(flag == 0)// For the right motor, inside track, ensures min turn speed is 190 (stop)
        final_speed = ((speed - *turn_spd_ptr + *r_correct_ptr) >= 190) ?
                        (speed - *turn_spd_ptr + *r_correct_ptr) : 190;
else if(flag == 1)// If right side it outside turn, max turn spd at 255 (full fwd)
        final_speed = ((speed + *turn_spd_ptr + *r_correct_ptr) <= 255) ?
                        (speed + *turn_spd_ptr + *r_correct_ptr) : 255;
}       else    // Left motor command, min turn speed is 64(stop) and 127 (full fwd)
{       if(flag == 0)// For the right motor, inside track, ensures min turn speed is 64 (stop)
        final_speed = ((speed - *turn_spd_ptr + *l_correct_ptr) >= 64) ?
                        (speed - *turn_spd_ptr  + *l_correct_ptr) : 64;
else if(flag == 1)// If right side it outside turn, max turn spd at 127 (full fwd)
        final_speed = ((speed + *turn_spd_ptr + *l_correct_ptr) <= 127) ?
                        (speed + *turn_spd_ptr + *l_correct_ptr) : 127;
} // end parsing if
```

```
        return final_speed;
} // end turn_simplified
```

C.8 Waypoint Control Node

Following code performs manual control of MONTe.

```
/***************************************************************************
Title:      MONTe_Waypoint_Control 0.0 (ROS Node)

Author:     Jason Hickle

Purpose:  Node that allows user to input waypoints over VNC server.
        Handles the following functions:
                1) Add waypoints to queue.
                2) Send route to New_Waypoints
                3) Delete all waypoints
                4) Update Command_Flags when in auto-nav.

Use:        Rudimentary user input program.  Inputs waypoints and then sends
        them to Waypoint_Processing via New_Waypoint_T. Route is sent at 4Hz
        to ensure waypoints are not lost. This synchs up with program rate.

          Utilizes class structure for manipulating route.

          Actions not used in current code, but kept for future use. Could be
        used to indicate search routines, return to base, etc.  Current
        behavior is to stop after completion of last waypoint.

          Next step is to take the class structure further and subsume
        ROS initilizations into the private portion.  This will tranisition
        to a more object-oriented format than currently implemented.

ROS Notes:
        Name-              "MONTe_Waypoint_Control"

        Publications-      "New_Waypoints_T"
                           "Command_Flags_T"

        Subscriptions-     None

        Messages-          Waypoint.msg
                           Command_Flags.msg

        Services-          None

Version History:

        ---- Version 0.0 ----
        May 19th, 2011
        LT Jason Hickle
```

```
        Allow input, deletion and sending of new waypoints.

Notes:  Further information can be found on ROS Wiki page:
                http://www.ros.org/wiki/
****************************************************************************/

/*      Libraries                       */
#include <ros/ros.h>
#include <signal.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>

#include "MONTe/Command_Flags.h"        // MONTe messages
#include "MONTe/Waypoint.h"

/*      Defines                         */
// Debugging options, uncomment to enable
#define WP_PRINT        // Print statement for new waypoints

#define KEYCODE_D 0x64  // For user inputs
#define KEYCODE_N 0x6e
#define KEYCODE_S 0x73

/*      WP stores all data necessary to navigate to, and determine          /
/       behavior mode for a route.                                          */
        typedef struct {
                int number;
                double latitude;
                double longitude;
                char action;
                } WP;

/*      class Waypoints provides methods for manipulating a waypoint route     */
        class Waypoints {
                private:

                public:
                        struct {
                                int number;
                                double latitude;
                                double longitude;
                                char action;
                        } wp[10];

                        int wp_count;
                        bool wp_entered;

                        Waypoints(){            // Constructor for class Waypoints
                                wp[0].number = 0;
                                wp_count = 0;
```

```
                                        wp_entered = 0; // Ensures route not sent until wp is entered
                                        }

                        void Add_Waypoint();
                        void Delete_Route();
                };

void Waypoints::Add_Waypoint()
{
        double temp_lat, temp_lon;
        char temp_action;

        wp[wp_count].number =  wp_count;

        printf("\nPlease enter waypoint %d latitude (dec degrees, N is positive): ", wp_count);
        scanf("%lf", &temp_lat);

        wp[wp_count].latitude =  temp_lat;

        printf("\nPlease enter waypoint %d longitude (dec degrees, E is positive): ", wp_count);
        scanf("%lf", &temp_lon);

        wp[wp_count].longitude =  temp_lon;

// Need to get better i/o option for following
/*      //do    {
                printf("\nPlease enter waypoint %d action (a-continue, s-stop): ",  wp_count);
                scanf("%c", &temp_action);
        //} while ((temp_action != 'a') || (temp_action != 's'));

        wp[wp_count].action =  temp_action; */

#ifdef WP_PRINT
ROS_INFO("Wp #%d, Latitude - %lf, Longitude = %lf", wp_count, wp[wp_count].latitude,
                        wp[wp_count].longitude);
#endif
        wp_count++;
        wp_entered = 1;
}

void Waypoints::Delete_Route() // Recursive function to delete stored wp data
{
        if(wp_count > 9)        // Prevents out of out of bounds
                wp_count = 9;

        while(wp_count > 0)     // Deletes all data until wp_count=0
        {
                wp[wp_count].number = 0;  // Null all data
                wp[wp_count].latitude = 0.0;
                wp[wp_count].longitude = 0.0;
                wp[wp_count].action = NULL;
```

```
            wp_count--;                // Decrement wp_count for next recursion
            Waypoints::Delete_Route();  // Call Delete_Route again to delete next wp
        }

        if(wp_count == 0) // Check in case of bad data
        {
            wp[wp_count].number = 0;  // Null all data
            wp[wp_count].latitude = 0.0;
            wp[wp_count].longitude = 0.0;
            wp[wp_count].action = NULL;

            wp_entered = 0; // Reset flag to indicate no stored waypoints
        }
}

/*      Global Variables        */

// Keyboard control variables
int kfd = 0;
struct termios cooked, raw;
int counter;

/*      Functional Prototypes   */
void Instructions();

void quit(int sig);

int main(int argc, char **argv)
{
/*      Initializations         */
        // Perform initializations for ROS
        ros::init(argc, argv, "MONTe_Waypoint_Control");  // Set up ROS node

        ros::NodeHandle n;  // set up handle for this node

        ros::Rate loop_rate(4);          // Sets 4hz cycle for main loop

        // Set up all publications for node
        ros::Publisher cmd_flg_pub = n.advertise<MONTe::Command_Flags>("Command_Flags_T", 1);
        ros::Publisher new_wp_pub = n.advertise<MONTe::Waypoint>("New_Waypoint_T", 10);

        MONTe::Command_Flags flags;
        MONTe::Waypoint new_wp;

        signal(SIGINT,quit);

        Waypoints Waypoint_Queue;

/*      Initialize keyboard for user input      */
                // get the console in raw mode
        tcgetattr(kfd, &cooked);
        memcpy(&raw, &cooked, sizeof(struct termios));
```

```
raw.c_lflag &=~ (ICANON | ECHO);

        // Setting a new line, then end of file
raw.c_cc[VEOL] = 1;
raw.c_cc[VEOF] = 2;
tcsetattr(kfd, TCSANOW, &raw);
while(ros::ok())
{
        char input_cmd;

        Instructions(); // Print instructions at each loop

// get the next event from the keyboard
        if(read(kfd, &input_cmd, 1) < 0)
        {
                perror("read():");
                exit(-1);
        }

        switch (input_cmd)
        {
                case KEYCODE_N: // Enter new waypoint
                        if(Waypoint_Queue.wp_count < 10)
                                Waypoint_Queue.Add_Waypoint();
                                else
                                        ROS_INFO("Max waypoints reached!");
                        break;

                case KEYCODE_S: // Send new route
                if((Waypoint_Queue.wp_entered == 1) && (Waypoint_Queue.wp_count > 0))
                        {
                                counter = 0; // Send waypoints

                        // Warn Waypoint_Processing new route incoming
                                flags.auto_nav = 0;
                                flags.nav_mode = 'A';
                                // Navigation takes over nav_mode
                                flags.incoming_route = 1;
                                //  Indicate new route to system
                                cmd_flg_pub.publish(flags);

                                loop_rate.sleep();      // Sleeps to maintain loop_rate

                //  Enter send waypoint loop, will send messages until all wp sent
                                while(counter < Waypoint_Queue.wp_count)
                                {
                                new_wp.latitude = Waypoint_Queue.wp[counter].latitude;
                                new_wp.longitude = Waypoint_Queue.wp[counter].longitude;
                                new_wp.action = 'a'; // stop and wait
                                new_wp.wp_num = counter;
                                new_wp.route = Waypoint_Queue.wp_count;
```

```
                                        new_wp_pub.publish(new_wp);
                                        // Publish to ROS Topic

                                        #ifdef WP_PRINT
                                        ROS_INFO("Waypoint %d sent!", counter);
                                        #endif

                                        flags.auto_nav = 1;
                                        //Place MONTe in autonomous navigation
                                        if(counter == 0)
                                                flags.nav_mode = 'N';
                                //Tell Waypoint_Processing to send first waypoint
                                //  for first waypoint sent
                                                else
                                                        flags.nav_mode = 'A';
                                        //Navigation takes over nav_mode
                                        flags.incoming_route = 1;
                                        //Indicate new route to system
                                        cmd_flg_pub.publish(flags);
                                        counter++;
                                        loop_rate.sleep();
                                        // Sleeps to maintain loop_rate
                                } // End send waypoint loop
                        }       else
                                {
                                        ROS_INFO("No route in queue to send.");
                                        continue;
                                }


                        ROS_INFO("MONTe is in autonomous navigation");

                        break;

                case KEYCODE_D: // Delete route
                        if(Waypoint_Queue.wp_entered == 1)
                                Waypoint_Queue.Delete_Route();
                                else
                                        ROS_INFO("No waypoints in queue to delete.");

                        break;

                default:
                        continue;
        }  // end switch
    }  // end main while loop

} //end main

/**********************************************************************
Function:       Instructions
```

```
Description:     Print out instructions for node

Parameter:       None

Return Value:    None
***************************************************************************/

void Instructions()
{
        printf("\nWelcome to MONTe waypoint control ...\n");
        printf("Please enter:\n\t\tn - Enter new waypoint\n\t\td - Delete route\n");
        printf("\t\ts - Send route\n");
}  // End Instructions

/***************************************************************************
Function: quit

Description:

Parameter:       1)

Return Value:
***************************************************************************/
void quit(int sig)
{
  tcsetattr(kfd, TCSANOW, &cooked);
  ros::shutdown();
  exit(0);
}
```

APPENDIX D:
ROS Messages

The following are the message formats used by MONTe.

```
# All messages for MONTe
# Used for sending information between topics

# Command_Flags.msg

# Contains all behavior flags for MONTe.

# Autonomous Navigation flag: 1 for autonav, 0 for manual control
bool auto_nav

# Navigation Mode: A - Enroute to current Waypoint, N - Current waypoint reached
char nav_mode

# Route flag indicates new set of waypoints
bool incoming_route


# Nav_Data.msg

# Send a waypoint for MONTe.

# Latitude in decimal degrees.
float64 latitude

# Longitude in decimal minutes
float64 longitude

# Heading
float64 heading

# Action character
char action


# Plant_Command.msg
#
# Basic message for manually controlling MONTe in simplified serial mode.

# Speed command for left motor. Range is 1(Full Reverse)-> 64 (Stop) <- 127 (Full Forward)
uint8 left

# Speed command for left motor. Range is 128(Full Reverse)-> 192 (Stop) <- 255 (Full Forward)
uint8 right
```

```
# Waypoint.msg
#
# Send a waypoint for MONTe.

# Latitude in decimal degrees.
float64 latitude

# Longitude in decimal minutes
float64 longitude

# Action character
char action

# Waypoint number (0-9)
int8 wp_num

# Number of waypoints in route (New_Waypoint only)
int8 route
```

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Physics Department
   Naval Postgraduate School
   Monterey, California

4. Dick Harkins
   Department of Applied Physics
   Naval Postgraduate School
   Monterey, California

5. Timothy Chung
   Department of Systems Engineering
   Naval Postgraduate School
   Monterey, California

6. Ravi Vaidyanathan
   Department of Systems Engineering
   Naval Postgraduate School
   Monterey, California